The University of Florida

Wireless Communications Dissertation Studies

Project Director: Dr. Fred J. Taylor

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	10/18/00	Final Report - 06/01/96-05/31/99

4. TITLE AND SUBTITLE
A Mobile Computing Environment

5. FUNDING NUMBERS
G#N00014-96-1-0976

6. AUTHOR(S) Dr. F.J. Taylor, Mr. Stuart Lopata, and Dr. Jonathan D. Mellott

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Florida, Division of Sponsored Research,
219 Grinter Hall, P.O. Box 115500
Gainesville, FL 32611 (352)392-1582

8. PERFORMING ORGANIZATION REPORT NUMBER
1 and final

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
Ballston Centre Tower One, 800 North Quincy Street,
Arlington, VA 22217-5660
Attn: Andre M. Von Tilborg, ONR 311

10. SPONSORING/MONITORING AGENCY REPORT NUMBERS

11. SUPPLEMENTARY NOTES
none

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for Public Release

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The objective of the Department of the Navy award was to facilitate and stimulate new research in the area of wireless communications. The Navy support was in the form of graduate student stipends to support the dissertation research of U.S. national students in this field, especially in the area of defining innovative solutions at the physical layer (hardware). These accomplishments include:

* Design and development of fast DSP-ASIC processors (commercial versions licensed through Philips Semiconductor)
* Design and development of low-power DSP ASICs.
* Design of digital radio subsystem and validation procedures (including the HSP43211, Intersil digital down converter).
* Extensive compute and simulation facilities including Synopsys, Altera, TI, Philips, and Mathworks.

14. SUBJECT TERMS
Architectures, digital signal processing, wireless communications

15. NUMBER OF PAGES
218

16. PRICE CODE
-0-

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI St. Z39-18
298-102

Final Report:
Department of the Navy
SAMAS No. 4505624

Dissertation Research Fellowship Program

OBJECTIVES

The objective of the Department of the Navy SAMAS No. 4505624 award was to facilitate and stimulate new research in the area of wireless communications. The Navy support was in the form of graduate student stipends to support the dissertation research of U.S. national students in this field. The study was motivated by the proposer's historical involvement in this field, especially in the area of defining innovative solutions at the physical layer (hardware). These accomplishments include:

- Design and development of fast DSP-ASIC processors (commercial versions licensed through Philips Semiconductor)
- Design and development of low-power DSP ASICs.
- Design of digital radio subsystem and validation procedures (including the HSP43211, Intersil digital down converter).
- Extensive compute and simulation facilities including Synopsys, Altera, TI, Philips, Mathworks, and other resources.

The reported dissertation studies made extensive use of all these resources.

ACCOMPLISHMENTS

Under the Department of the Navy SAMAS No. 4505624 award, one Ph.D. dissertation was completed and a second is nearing completion. They are:

1. Very Long Instruction Word Architecture for Digital Signal Processing, Dr. Jonathon Mellott.

2. Wireless Local Area Networking and Channeling at 2.4GHz, Stuart Lopata

The completed dissertation is reported in abstract form, and in complete form as an attachment. The on-going dissertation research is abstracted and included in interim report form as an attachment.

Very Long Instruction Word Architecture for Digital Signal Processing

Dr. Jonathon D. Mellott (Ph.D., 1997, University of Florida)

The research reported resulted in a Ph.D. degree awarded to Jonathon D. Mellott for his original research in Very Long Instruction Word Architectures (VLIW) for digital signal processing (DSP). Under the Department of the Navy SAMAS No. 4505624 support, Dr. Mellott developed a new class of processor that is applicable to a number of wireless information communication applications. The study was motivated by the new class of VLIW architectures being promoted by Texas Instruments (TI) and a knowledge of superior (speed and power) arithmetic structure for this class of processor. The research, reported as an attachment, indicates that significant benefits can be gained in both speed and power using this new architecture.

Scholarly paper resulting from the dissertation research are as follows:

1. Mellott, J., Lewis, M., and Taylor, F., "ASAP - A 2D DFT VLSI Processor and Architecture," IEEE ICASSP Conference, Atlanta, 1996.

2. Mellott, J., and Taylor, F., "Very Long Instruction Word Architecture for DSP," IEEE ICASSP Conference, Munich, 1997.

3. Meyer-Baese, U., Mellott, J., and Taylor, F., "Frequency Sampling Filter Bank Using the RNS," IEEE ICASSP Conference, Munich, 1997.

4. Taylor, F., and Mellott, J., Hands On Digital Signal Processing, McGraw Hill, 1998.

Upon graduation, Dr. Mellott joined the Athena Group Inc. as a lead VLSI engineer where he has led design activities involving VLIW architectures under the sponsorship of NIST, BMDO, and the Army. The NIST project lead to a technology that is being marketed by Philips Semiconductor for ASIC applications.

Wireless Local Area Networking and Channeling at 2.4GHz

Stuart Lopata (Ph.D. expected in 2001, University of Florida)

The reported research provides a foundation for Mr. Lopata's dissertation research into IEEE802.11 class wireless LAN (WLAN) communications. The research conducted under the Department of the Navy SAMAS No. 4505624 project involved making detained measurement of the 2.4GHz IEEE802.11 class signal environment. This data did not exist and is essential to properly direct the dissertation research. Measurements included signal strengths, noise level, interference sources, multi-path effects, and antenna sensitivity. This information is now used to "seed" both the mathematical models and simulators used in the research of the enhanced 5GHz IEEE 802.11 WLAN that is based on an orthogonal frequency division modulation (OFDM) standard.

Publications, Stuart Lopata author

Optimum Frequency Estimation Strategies for Repeated Training Signals via Efficient Time-Domain Processing, in review, IEEE Transactions of Communications

OFDM Channel Estimation in an Indoor Wireless Environment, in review, IEEE Transactions of Communications

Multiple Threshold Detection and Timing Estimation in OFDM for WLANs, in review, IEEE Transactions of Communications

Appendix A: Very Long Instruction Word Architecture for Digital Signal Processing

VERY LONG INSTRUCTION WORD ARCHITECTURES FOR DIGITAL SIGNAL PROCESSING

By

JONATHON D. MELLOTT

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1997

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

VERY LONG INSTRUCTION WORD ARCHITECTURES FOR DIGITAL SIGNAL PROCESSING

By

Jonathon D. Mellott

December, 1997

Chairman: Dr. Fred J. Taylor
Major Department: Electrical and Computer Engineering

Very long instruction word (VLIW) architecture offers an opportunity for superior multiprocessor digital signal processor implementations. By eschewing the hardware resource management provided in superscalar and superpipelined processor implementations, a VLIW processor has more available hardware resources for computations. The disadvantages of the VLIW approach are that object code is no longer compatible across multiple generations of processors and that the compilation technology to support a VLIW processor is more complicated than that required by traditional processor architectures.

This dissertation describes a VLIW architecture for digital signal processing. The described architecture has multiple functional units, including a residue number system convolution processor. The convolution processor is based upon the Athena sensor arithmetic processor, a 1.2 billion operation per second SIMD convolution processor, which is also described. To solve the difficulties associated with software development for a VLIW digital signal processing microprocessor, a new high-level language based upon the C programming language is described. Implementations of several key digital signal processing algorithms are analyzed with respect to opportu-

nities for instruction-level and block-level parallelism, and their resource requirements in the context of a VLIW digital signal processing environment.

CHAPTER 1
INTRODUCTION

> When I used to build racing engines a few decades ago, we had someone stuff a 500 HP street racing engine in a Ford Falcon. It turned the tires but didn't put the power on the ground. Bigger tires were added so it got enough bite to break the suspension, which was improved until the transmission and driveshaft broke, which were upgraded so the last version worked really well and twisted the frame so badly the windshield popped out on the first shift, and the doors wouldn't ever close once opened.
> —*Bill Davidsen*

Since the 1970s the semiconductor industry has experienced geometric growth in the number of transistors that can be placed on a chip [1], see Figure 1.1. With time, designers of digital signal processing (DSP) devices have been able to take advantage of the geometric growth with respect to the number of transistors that could be placed on a chip to produce successive generations of processors that offered greater performance due to the increased number of circuit elements available. For example, consider the Texas Instruments TMS320 DSP family. Using the sixteen bit, fixed-point C10 generation as a baseline, the C20 generation augmented the C10 architecture with a fast multiplier. The C30 generation used a thirty-two bit floating point architecture. The C40 generation added DMA (direct memory access) processors for multicomputer interconnect to the C30 core. As the number of available circuit elements per chip increases, increasingly more functionality can be added. As the number of functional units that can be placed on a single chip processor increases the question of how to actually use those resources becomes very difficult to answer.

In this chapter motivation will be offered for research into the insertion of very long instruction word techniques into the design of architectures for high performance digital signal processors that are not highly application specific. Existing solutions based on general purpose digital signal processors have concentrated on multiple-instruction, multiple-data (MIMD) parallel solutions (such as Texas Instruments TMS320C40 and TMS320C80 products). These solutions have not proven to be entirely satisfactory due to system integration and software development obstacles. Digital signal processing applications are especially well suited for VLIW architectures, and the nature of digital signal processing implementations sidesteps the most troublesome software lifecycle compatibility issues that currently hinder the widespread application of VLIW techniques in the general purpose computer market.

VLIW techniques can be used to exploit opportunities for instruction level parallelism just as superscalar and superpipelining techniques are also used to exploit opportunities for instruction level parallelism. VLIW instruction scheduling techniques can also be adapted to allow opportunities for block level parallelism to be exploited. A final advantage of VLIW architecture over the competing superscalar architecture is that the hardware resources that are expended in superscalar architectures to support multiple instruction issue are eliminated in VLIW and are therefore available for additional functional units or other architectural resources.

This introduction is organized as follows. The first section provides a comparison of general purpose processors versus DSP processors. This is necessary to justify some of the assumptions under which the work has proceeded. The next section provides motivation for VLIW insertion into digital signal processors by examining the characteristics of digital signal processing algorithms and the architectural resources necessary to perform those algorithms efficiently, and provides a survey of available

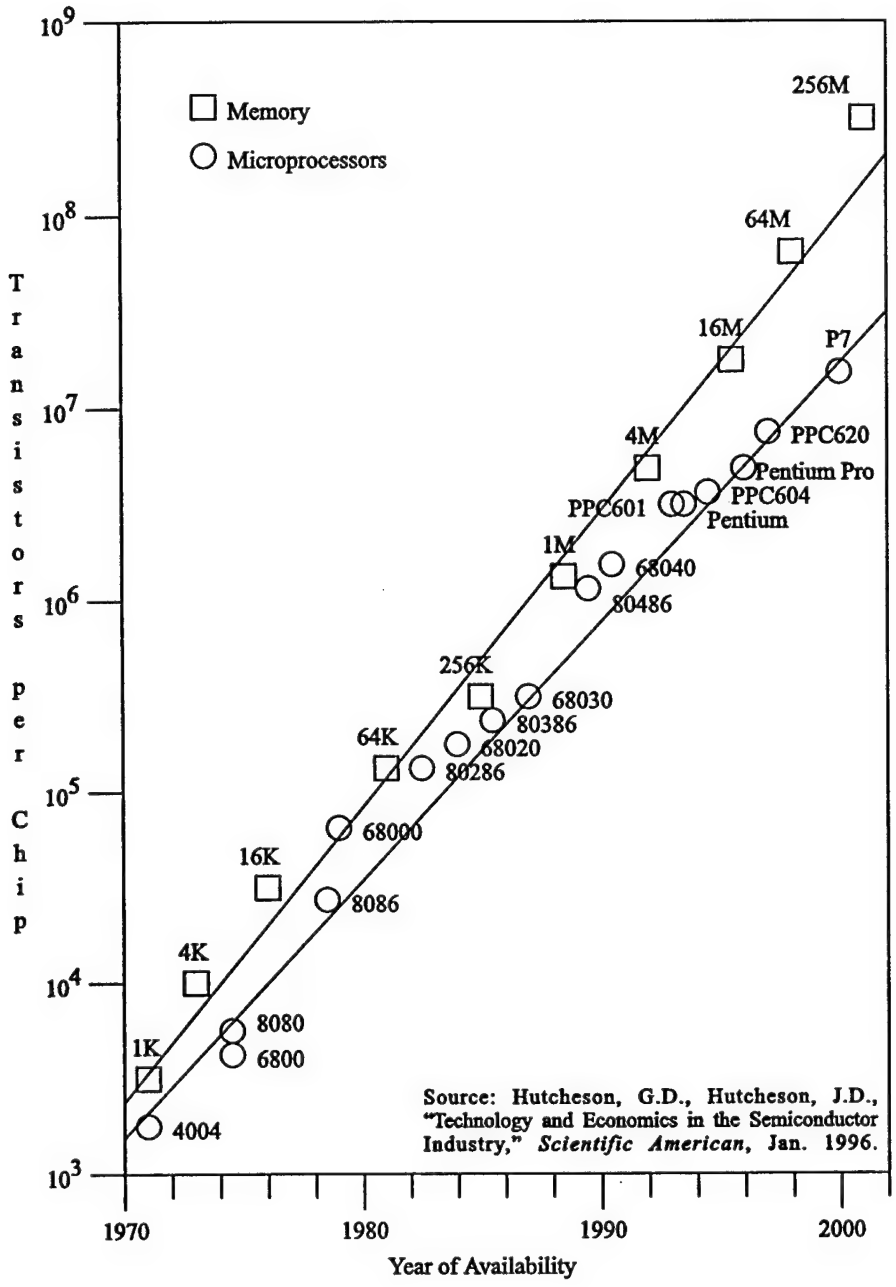


Figure 1.1: Transistor Densities per Chip Trends for Memories and Microprocessors

techniques for exploiting instruction-level parallelism. The final section introduces the research reported in this dissertation.

1.1 Comparison of General Purpose Processors Versus DSP Processors

To develop the motivation for this work it is necessary to understand the differences between general purpose processors and digital signal processors. General purpose processors are defined as those processors designed to execute a variety of algorithms efficiently. Features found on most general purpose processors (although not all) include

- multiple data types supported by the processor hardware,
- multi-level cache memories,
- paged virtual memory management in hardware,
- support for hardware context management including supervisor and user modes,
- unpredictable instruction execution timing,
- large general purpose register files,
- orthogonal instruction sets, and
- simple or complex memory addressing depending upon whether the processor is a RISC (reduced instruction set computer) or CISC (complex instruction set computer).

The most important data types for general purpose processors are the character type followed by the integer type. From the viewpoint of market share, the majority of general purpose processors will be employed in business applications that involve text and database processing. Floating-point arithmetic is generally not crucial in

most applications run on general purpose computers, although there are niche markets where this is not true (e.g., the technical workstation market).

Cache memories have been demonstrated to be a useful enhancement for many general purpose processors due to demonstrated instruction locality and data locality for many types of problems run on general purpose computers. The inclusion of sometimes substantial cache memories in general purpose computers is made on the assumption that programs that demonstrate instruction or data locality will be run on that computer. This assumption will hereafter be referred to as the "cache assumption." Frequently, the cache assumption is used to justify the design of shared memory multiprocessing general purpose computers where the main memory is connected to the processors via a shared bus. If the cache assumption is violated the performance of single and multiprocessing general purpose computers is generally degraded. The types of applications run on classic vector supercomputers, such as the various Cray implementations, were assumed by their designers to violate the cache assumption for data access and therefore eschew data caches [2].

Large register files are included in many general purpose architectures, although there are exceptions (e.g., the Intel x86). Since most general purpose machines operate on scalar data and the cache assumption usually holds, large register files are generally beneficial. General purpose registers and orthogonal instruction sets tend to make it easier to write compilers that emit efficient object code, and are also beneficial to the assembly language programmer. Also, the load-store architectural constraint used in many RISC processors makes larger register files attractive: since external memory can only be accessed by load and store operations it is desirable to keep more operands on hand in the register file to obtain good performance.

Hardware support for the management of virtual memory and multiple process contexts is desirable in general purpose computers. Most general purpose processors

support timeshared execution of multiple processes; even single-user desktop computers generally are running many processes simultaneously. Virtual memory allows programs to run in a degraded manner if their primary memory requirements exceed available resources. The penalty for virtual memory is increased data access latency due to address translation penalties and long page fault latencies. The latter is generally managed by switching the processor context to another process so that the processor does not idle while a page fault is being serviced. Support for multiple process contexts by a general purpose computer is therefore crucial for optimal use of the processor resource among multiple tasks.

Instruction execution timing on general purpose processors is generally unpredictable: this is a result of a myriad of features designed to enhance the performance of the processor. Cache memory and virtual memory introduce a substantial amount of uncertainty in instruction execution timing. The amount of time required to read or write a particular location in memory will depend upon whether or not a cache hit occurs, at which level of the cache it hits, whether or not that virtual address resides in the TLB (translation look-aside buffer), the latency of the main memory which can be affected by when the last access occurred and refresh requirements in addition to access contention by other processors or DMA. Various architectural enhancements such as superscalar execution, speculative execution, out-of-order execution, and branch target caches may further confound any attempt to measure the execution time of an instruction.

A derivative class of general purpose processor is the microcontroller. Most microcontrollers are derived from successful general purpose microprocessor designs, although some are original designs. Microcontrollers are typically targeted at embedded applications like many digital signal processors, but typically these applications do not require the arithmetic performance of the digital signal processor. Microcon-

trollers usually eliminate features such as large cache memories and virtual memory and instead add integrated peripheral interfaces to support the intended embedded applications.

In contrast to general purpose processors, digital signal processors are designed primarily to do arithmetic very efficiently. Most digital signal processing applications are embedded and hard real-time in nature. Additional architectural features are added so as to enhance the execution of typical digital signal processing algorithms. DSP processors are typified by the following characteristics:

- only one or two data types supported by the processor hardware,
- no data cache memory,
- no memory management hardware,
- no support for hardware context management,
- exposed pipelines,
- predictable instruction execution timing,
- limited register files with special purpose registers,
- non-orthogonal instruction sets,
- enhanced memory addressing modes,
- on-board fast RAM (random access memory) and/or ROM (read-only memory), and
- on-board DMA.

Most DSP processors only support one data type really well. Other data types, if supported, usually only have partial support. Since the primary purpose of a

digital signal processor is to perform arithmetic, elimination of excess data types is a reasonable optimization. This optimization extends to the datapaths; dynamic bus sizing and fractional word width operations are generally eliminated.

Digital signal processing applications usually have hard real-time requirements that dictate that instruction execution timing be predictable. The cache assumption may also be violated for data access, depending upon the problem to be solved with the digital signal processor. Therefore data cache memory is generally not included in digital signal processors. Like the classic vector supercomputers, most digital signal processors instead devote resources to on-chip fast RAM or ROM that is explicitly managed by the programmer. Many of the same justifications used to justify the vector registers in vector machines can be used to justify on-chip memories found in digital signal processors. From a VLSI manufacturing perspective, on-chip memories produce an excellent return on investment since there are many well understood techniques to enhance the manufacturing yield of memories [3]. While the same arguments are made for on-chip cache memories, the return on investment is greater for on-chip RAM (assuming that it is used efficiently) since tag RAM and address comparators required for cache implementations are eliminated, a substantial savings that also results in reduced access latency when compared with cache memory.

Memory management hardware is not included in digital signal processors since virtual memory cannot be implemented in systems with hard real-time requirements. When secondary storage is required for the processing of data, that storage is generally managed by the programmer. Likewise, most digital signal processors are dedicated to single problems and therefore do not need process-level multitasking, so hardware context management is not required. When multitasking is required on a limited scale it can be provided through cooperative means or via device driven interrupts.

Digital signal processors typically operate on arrays of data rather than perform operations on scalars and therefore do not gain great benefit from large register files. Register files in digital signal processors typically feature a number of special purpose registers to support exotic addressing modes and other features that could not be justified in general purpose processors but are well used in digital signal processing applications. Consequently, instruction sets for digital signal processors are typically non-orthogonal. Most high-level language compilers for digital signal processors are not able to take advantage of many of the special features of the digital signal processor's instruction set architecture and therefore do not emit efficient object code. This is largely due to the fact that most high-level languages are designed for general purpose processors. Consequently, most high-level language development environments for digital signal processors rely upon libraries of hand-coded subprograms to achieve adequate performance. Programs that are fully hand-coded in assembly language can usually achieve even better performance than that obtainable with optimized library code. The reliance upon hand optimized assembler code for digital signal processing applications has historically been a reasonable approach: since most DSP products represent a combination of hardware and software, the increased life-cycle costs of assembler code can be recovered via the reduction in per-unit hardware costs of the components. Given this paradigm, instruction set orthogonality is sacrificed to add special features that benefit DSP applications. The assembler programmer is able to take advantage of those special features that would not be used by a standard high-level language compiler.

1.2 Motivation for VLIW Insertion in Digital Signal Processors

This section will describe the motivation for the study of VLIW insertion into digital signal processors. This will be done by first examining the characteristics of a large

class of digital signal processing algorithms and from those characteristics extracting architectural features needed to support digital signal processing. Opportunities for instruction level parallelism will also be identified. Finally the motivation for examining VLIW versus other throughput enhancement techniques will be examined.

1.2.1 Characteristics of digital signal processing algorithms

Most digital signal processing algorithms are dominated by multiply-accumulate operations used to form sums of products [4]. Existing digital signal processors are optimized to compute expressions of the form

$$z = \sum_i x_i y_i. \tag{1.1}$$

For example, the finite impulse response (FIR) filter is computed using

$$y_n = \sum_{i=0}^{N-1} a_i x_{n-i}, \tag{1.2}$$

where the finite sequence $\{a_i\}$ is the set of filter coefficients, $\{x_i\}$ is the input sequence, and $\{y_i\}$ is the output sequence. The form of (1.2) is that of the discrete convolution operation. From the perspective of digital signal processors the discrete correlation and convolution are essentially equivalent: they differ only in the ordering of a set of coefficients.

Another type of common operation performed in digital signal processing is vector arithmetic. In particular, the sum of two vectors $\{x_n\}$ and $\{y_n\}$ given as $z_n = x_n + y_n$ is used to superimpose or add signals. Alternatively, this sum may take the form $z_n = \alpha x_n + y_n$ where α is a a scalar. This form is sometimes called a SAXPY (Scalar α X Plus Y) operation [5]. The product of two vectors $\{x_n\}$ and $\{y_n\}$, given

as $z_n = x_n y_n$, is used in windowing operations commonly found in basic spectral estimation applications [6].

The discrete Fourier transform (DFT) is very important in applications ranging from spectral estimation to automatic target recognition. The DFT of a finite sequence $\{x_n\}$ of length N is given as

$$X_n = \sum_{k=0}^{N-1} x_n e^{-j2\pi nk/N}, \tag{1.3}$$

for $n \in \{0, 1, 2, \ldots, N-1\}$. The DFT as given in (1.3) is an expensive operation to perform requiring N^2 complex multiply-accumulates to compute X_n for all $n \in \{0, 1, 2, \ldots, N-1\}$. Alternate means of computation of DFTs have been developed to reduce the computational expense of the DFT or to gain an implementation advantage [7]. The Cooley-Tukey fast Fourier transform (FFT) [8] and the Good-Thomas fast Fourier transform [9] both reduce the complexity of the transform to approximately $O(N \log N)$ operations. The Cooley-Tukey FFT can be constructed using *bit-reversed* addressing, a feature most digital signal processors have included, thus making the Cooley-Tukey FFT the most popular implementation. The Good-Thomas FFT is composed of many small prime block length DFTs. It has been demonstrated to be advantageous in a VLSI sense to use the Good-Thomas FFT rather than the Cooley-Tukey FFT as the required small prime block length DFTs can be efficiently performed using dedicated VLSI hardware [10, 11, 12, 13, 14]. Despite its advantages in VLSI hardware, the Good-Thomas FFT has seen only limited application.

1.2.2 Architectural resources for digital signal processing

Digital signal processors are designed around a different set of assumptions than those which drive the design of general purpose processors. First, digital signal pro-

cessors generally operate on arrays of data rather than scalars so the scalar load-store architectures found in general purpose RISCs don't make a lot of sense. The economics of software development for digital signal processors is different from that for general purpose applications. Digital signal processing problems tend to be algorithmically smaller than a word processor, for example. In many cases the ability to use a slower and therefore cheaper digital signal processor by expending some additional software engineering effort is economically attractive: a good return on investment may be achieved if five dollars per unit of manufacturing cost can be saved in a product that will ship a million units by expending an extra man-year of development effort. A consequence of these factors is that most programming of digital signal processors is done in assembly language rather than high-level languages. In fact, digital signal processors have been architected to allow optimal assembly code to be written quickly to the point that compilers for *standard* high-level languages are unable to produce efficient code. This is essentially the CISC instruction set architecture paradigm.

Addressing modes. General purpose processors have either many addressing modes (CISC processors) or few addressing modes (RISC processors). CISC processors may support addressing modes such as direct, register or memory indirect, indirect indexed, indirect with displacement, indirect indexed with displacement, and the indexed modes may support pre- and post- increment or decrement of the indices. Historically, complex addressing modes have resulted in higher code entropy which has two consequences: first, the productivity of the assembly language programmer is enhanced, and second, the resulting object code is more compact. A number of factors have contributed to the disappearance of complex addressing modes characteristic of CISC processors. The first is the change in the economics of hardware

costs versus software development costs: thirty years ago software development was cheap and hardware was expensive so hand coded assembler was commonly used in application programs, while today hardware is inexpensive relative to software development costs so most applications are coded exclusively in high-level languages. Another issue is related to the first: it has proven to be difficult to get compilers to take full advantage of complicated addressing modes and non-orthogonal instruction sets. Another strike against complex addressing modes in general purpose computers is that the complex addressing modes tend to cause pipeline stalls due to the complicated data dependencies produced by the complex addressing modes. Even modern CISC implementations have been optimized so that better performance results when complex addressing modes are avoided. Eschewing complex addressing modes has led to the adoption of a load-store philosophy that allows functional units to accept issues without stalling due to depending upon data stored in memory. By moving to a register indirect load-store architecture all of the more complex addressing operations are performed in software thus allowing greater flexibility in scheduling instruction issue. A register indirect load-store architecture synthesizes more complicated "addressing modes" with several simple instructions. The compiler is free to statically arrange these instructions with awareness of the impact of adjacent instructions on the scheduling of processor resources. The processor may also elect to rearrange the execution of these simple instructions within the constraints of available resources and apparent data dependencies. In contrast the classic CISC has the micro-operations of each instruction statically scheduled in the microprogram for a particular instruction.

Digital signal processing applications frequently require non-sequential access to data arrays using modular or bit reversed addressing. These addressing modes are not easily supported in general purpose RISC or CISC processors. For maximum performance in digital signal processing applications it is sensible to add dedicated

hardware support for these addressing modes. To summarize, the addressing modes required include

- address register indirect,
- address register indirect with unit stride and non-unit stride modular indexing, and
- address register indirect with bit-reversed indexing.

Existing digital signal processor architectures are single-issue so, with the exception of the special modes indicated, the address register file and arithmetic unit would be similar to that found general purpose architectures. To support multiple issue it will be necessary to define either a hardware or software mechanism to support concurrent address generation for multiple function units. How to do this efficiently, and whether it should be done at the hardware or software (or both) level, is an open question.

Instruction set enhancements. Since execution time in digital signal processing applications is dominated by operations of the form in Equation 1.1 it is sensible to provide instruction set support for executing a loop a fixed number of times. In fact looping based upon the value of a counter is the most common branching operation in digital signal processors; so much so that many have dedicated instructions to implement zero or reduced penalty looping. For example, both the Texas Instruments TMS320 series processors and Motorola DSP56000 series processors support an instruction that causes the next machine instruction to be repeated a fixed number of times [15, 16]. Consequently, a justification for dedicating substantial resources to a branch-target cache [17, 18] cannot be found. Large scale branch-target caching

makes more sense in general purpose applications as many of these applications have branching patterns that are difficult to predict at compile time.

Most integer digital signal processors actually employ a fixed-point arithmetic format. The fixed-point format is achieved by integrating shifters with the multiplier-accumulator so as to allow pipelined adjustment of operands and results. The multipliers and accumulators included in most fixed-point digital signal processors are oversized to allow transient computations to exceed the normal word width of the processor. For example, the Texas Instruments TMS320C50, a sixteen bit fixed-point digital signal processor, has a multiplier that takes two sixteen bit operands and produces a thirty-two bit output as well as a thirty-two bit accumulator. Likewise the Motorola DSP56000, a twenty-four bit fixed-point digital signal processor, has a multiplier that takes twenty-four bit operands, produces forty-eight bit outputs and has a fifty-six bit accumulator. These processors include architectural support for controlling rounding and normalization of results.

Since the dominant language for programming digital signal processors is assembly, generally there is some effort put into the instruction set so as to make it easier for the assembly language programmer. For example, exposed pipelines are usually avoided, however, in the quest for higher performance at lower unit cost, designers of some processors (such as the TMS320C50) have resorted to exposed pipelines. Since multiply-accumulate operations are so common in digital signal processing applications, explicit multiply-accumulate instructions are included in the instruction set of digital signal processors. In general purpose processors multiply-accumulate operations might be supported via chaining of the multiplier and adder functional units (particularly in RISC implementations). Even the paragon of CISC processors, the VAX, didn't have an explicit floating-point multiply-accumulate instruction, al-

though it did have an extended integer multiply instruction that could be used in some instances to perform multiply-accumulate operations [19].

The problem of exposed pipelines is significant in that it hampers the assembly language programmer. It promises to become much worse in future processors as throughput considerations demand deeper pipelines. This is clearly a problem that must be managed in the programming tools. In particular, new programming tools should be able to migrate code among successive generations of processors with differing pipeline depths without programmer intervention.

Dataflow support. Since digital signal processors are designed to support real-time processing of large quantities of sampled data they generally have support for enhanced dataflow. Modified Harvard architectures are generally applied, particularly with respect to on-board memories. Some digital signal processors also support modified asymmetric Harvard architectures with respect to external interfaces that support data storage and I/O (input/output) operations. For example, the TMS320C30 has a twenty-four bit (16M word) primary addressing space for programs and data and a second thirteen bit (8K word) addressing space for data storage and peripherals.

Some digital signal processors include DMA controllers that are capable of performing memory-memory move operations concurrent with computational tasks. An independent DMA controller would typically be used to load new data into the on-chip memory while some computation is performed. This allows an internal Harvard architecture to be better exploited by keeping the processor busy with computation rather than programmed data I/O. Currently these DMA resources are managed explicitly by the programmer. To support rapid code development and portability it is important that the management of the DMA resources be simplified, at least, if not moved completely into the programming tools.

1.2.3 Techniques for exploiting instruction level parallelism

As VLSI (very large scale integration) technology has improved it has become possible to include additional hardware resources to enhance the performance of general purpose and application specific processors. To increase throughput in traditional von Neumann machines additional hardware resources are added to exploit opportunities for instruction level parallelism [20]. The techniques that have been developed to exploit opportunities for instruction level parallelism are superpipelining, superscalar architecture, dataflow processors and very long instruction word architecture. Since software development costs have spiraled upwards, a significant amount of work has been done in the area of automatic compiler-based optimization of high-level language code. To a certain extent the ability to automatically optimize code drives general purpose processor architecture. An excellent survey on the topic of compilation for parallel machines can be found in Gokhale and Carlson [21].

The technique of superpipelining has been exploited by processors such as the DEC Alpha and Intel Pentium Pro to achieve high throughput. Superpipelining works by adding pipeline stages so as to achieve a very short machine cycle thus allowing a high issue rate. While instructions are issued sequentially at a high rate they take many cycles to complete, so while one instruction is started several or many previous instructions may be in various stages of completion. The disadvantage of superpipelining is that it increases latency (the time from when an instruction is issued to when it is completed) and makes pipeline flushes more expensive. From a hardware perspective the addition of pipeline registers requires significant extra hardware resources. To hide the pipeline from the programmer and/or compiler the processor must keep track of resources that have been committed to instructions that are in progress in the pipeline. If resource conflicts occur then the pipeline is stalled or bubbles are introduced into the pipeline. Instructions are generally ordered by

the compiler, programmer, and/or processor so as to avoid pipeline stalling whenever possible.

Superscalar processors use multiple functional units to achieve instruction level parallelism. Examples of modern superscalar processors are the Intel Pentium which has two integer pipelines and one floating-point pipeline and the Sun/Texas Instruments SuperSPARC [22] which also has two integer pipelines and one floating-point pipeline. A high instruction issue rate is achieved by issuing more than one instruction per machine cycle. To do this the processor must track the resource requirements of each instruction to be sure that it does not conflict with resource requirements of instructions executing on other pipelines. Like superpipelined processors, superscalar processors rely on the programmer or compiler to arrange instructions so as to minimize resource conflicts and thereby maximize the instruction issue rate. Some recent processor implementations are capable of changing the order of execution (out-of-order execution) so as to optimize instruction issue, however, this technique is very hardware intensive.

Dataflow processors work by having each instruction indicate which subsequent instructions depend upon the results of a particular instruction. With this explicit dependence information encoded into the instruction stream it is relatively easy to issue instructions so as to achieve an optimal issue rate. Dataflow processors have not found success in the mainstream of general purpose processors but rather in research and application specific processors [23].

Superscalar and superpipelining are the current commercially dominant techniques for achieving instruction level parallelism. There has only been one significant commercial VLIW computer, the Trace Multiflow [24, 25]. There is also a research VLIW in recent literature, the VIPER [26, 27]. VLIW is similar to superscalar in that it depends upon multiple function units operating in parallel. VLIW differs

from superscalar in that it uses a very long instruction word to issue an instruction to each function unit on every instruction cycle. The resource interlocks that exist in superscalar processors to prevent resource conflicts are eschewed in VLIW machines in favor of resolving resource conflicts at compile or load time. In many ways this philosophy is similar to a microprogrammed controller with multiple functional units [28] such as some systems constructed using bit-slice devices [29]. The proponents of VLIW propose that the resources expended in superscalar processors to prevent resource conflicts are better spent on additional function units and that instruction scheduling is better performed in software rather than hardware, particularly since software instruction scheduling (by the compiler) can take advantage of a global view of the program as well as additional information that exists at the source code level but does not exist at the object code level. The early superscalar implementations were fairly successful in achieving good issue rate performance versus the number of function units. Later implementations have been somewhat less successful at maintaining function unit utilization; as the number of function units has increased issue rate efficiencies have decreased. For example, some new four-way superscalar implementations rarely achieve four issues per cycle. As the number of function units increases the difficulties in managing the units to achieve multiple issue is becoming increasingly complex. These problems are combining to motivate commercial interests to look at VLIW for next generation processor architectures. For example Intel and Hewlett-Packard are collaborating on a VLIW influenced processor to replace their existing x86 and PA-RISC products [30]. Significant obstacles — particularly software compatibility issues — remain to be solved in order for VLIW to significantly impact the general purpose computer market [20].

1.2.4 VLIW for digital signal processing

VLIW architecture insertion into digital signal processors is attractive for a variety of reasons. VLIW allows multiple function units to be used in a digital signal processor without the hardware overhead and cost associated with the bookkeeping functions, such as scoreboards [2], found in superscalar processors. The tradeoff is more complicated software development, however this is mitigated somewhat by most digital signal processing applications having relatively simple codes with limited flow control. This complexity can be managed with programming tools and these more advanced programming tools can be leveraged to allow selection of a particular VLIW architecture based upon programming requirements.

In addition to instruction level parallelism, digital signal processing codes frequently have opportunities for block level parallelism that can be exploited on VLIW processors [31]. For example, a windowing operation might precede a Fourier transform in a real-time spectrum analyzer. Using block level parallelism a VLIW processor might be windowing record $N+1$ while computing the Fourier transform of record N. The problem of code expansion is one that must be seriously considered in digital signal processing applications since memory for firmware is an expensive resource. However, it is worth noting that digital signal processing applications tend not to have a lot of flow control operations besides looping and thus do not exacerbate the code expansion problem [21].

A significant advantage of VLIW over superscalar implementations is predictable instruction execution timing. Since operations on a VLIW are scheduled into instructions at compile time and all pipeline stalls and bubbles are visible, execution time is easily determined. Another advantage is that the DSP developer is more tolerant of the software compatibility issues that currently hinder the application of VLIW to general purpose markets. In particular, the DSP developer tends to be much more

tolerant of the expense of retargeting application codes to different processor architectures: binary object code compatibility among different generations of processors is not required.

1.3 Research Activities

VLIW is an attractive approach for achieving parallelism, both instruction level and block level, for digital signal processing applications. Since digital signal processing applications are frequently very cost sensitive with respect to hardware, the cost benefits of VLIW are particularly attractive. Despite the obvious benefits of VLIW for digital signal processing there has been little interest in VLIW in the digital signal processing community, until recently. From a commercial perspective, attempts at parallelism for digital signal processing have relied upon expensive multiprocessor communications in Kung's Warp [32] and iWarp [33] processors, and Texas Instruments' TMS320C40 [34], or alternatively multiple independently programmed ALUs (arithmetic logic units) in Texas Instruments' TMS320C80 [35]. The iWarp processor was developed for commercial implementation by Intel but never sold in any volume. The TI TMS320C40 is essentially a TMS320 family floating-point processor with six integrated communications ports allowing C40 to C40 data I/O. Unfortunately the C40 has limited appeal due to high cost — largely driven by the die area overhead of the communication ports and the 391 pin interstitial ceramic PGA (pin grid array) package. Another impedance to widespread use of the C40 was the difficulty in writing code that used the C40's communications features. The new TMS320C80 combines a RISC floating-point processor with multiple ALUs under independent program control. The ALUs are optimized for pixel processing operations and the device is optimized for, and being marketed towards video processing applications.

Attempts to construct parallel processing digital signal processors have not been entirely successful. The currently extant commercial solutions rely on MIMD architectures that have proven to be difficult to use effectively for digital signal processing applications. The ultimate goal of this research was to arrive at a VLIW digital signal processor architecture that integrates RNS (residue number system) arithmetic elements into an architecture capable of performing general signal processing tasks. Inclusion of RNS processors is motivated by the high arithmetic bandwidth relative to die area that can be achieved with RNS. To achieve this goal, the constraints of digital signal processing applications and their differences from general purpose applications must be carefully considered. To this end, the following research activities were undertaken:

1. Identify the function units required of a VLIW digital signal processor. Quantify the number of units required, on-board memory requirements, and I/O bandwidth requirements. This will be driven by algorithmic requirements. This is not a quest for a single solution but rather a spectrum of solutions.

2. Study the insertion of RNS processing elements into a non-application specific VLIW digital signal processor. RNS has proven to be very attractive for application specific digital signal processors, however, it is difficult to use for non-application specific digital signal processors. Identify those elements required to integrate RNS computing with general DSP problems. Quantify advantages and disadvantages of RNS insertion.

The first objective is analytical in nature. While basic digital signal processing algorithms such as filtering and Fourier transforms are considered, more complex signal processing algorithms such as the QR decomposition [5] necessary for applications such as beamforming [36] are examined. To fully take advantage of the proposed ar-

chitecture it is necessary to identify where and under what constraints RNS processing can be applied as previously suggested by the author [37]. A spectrum of resource requirements are developed. This is consistent with the current trend towards processor cores with variants designed for specific markets; microprocessors of fifteen years ago may have only been offered in one or two variants whereas modern processors are offered in many tens of variants (hundreds or thousands if standard cell cores are included) [1].

The second objective is primarily a synthesis and comparative computer architecture problem. Since the developed architecture includes both RNS and conventional arithmetic elements, a balance is identified within realistic current and anticipated future technological constraints.

To tie these research objectives together it was necessary to address the problem of programming a full VLIW DSP microprocessor. The instruction scheduling problem is relatively well-understood and is a subject of ongoing research. This problem is not addressed here. To enable a program first, select hardware last system integration paradigm it is necessary to enable processor independent software development. The solution to this problem is to adopt a high-level language for programming DSP applications. Existing high-level languages are intended for general purpose computers, not digital signal processors. Therefore, the obvious conclusion is that a high-level language for digital signal processors and digital signal processing applications is required. The C programming language is considered a "high-level assembly language" for general purpose processors, however, performs poorly in digital signal processing applications, especially on DSP microprocessors.

To produce a high-level assembly language that works well for digital signal processing applications and DSP microprocessors, the semantics of the C programming language have been significantly modified, creating a new language, C_{DSP} . The

C_{DSP} language is an innovative approach to high-level language DSP programming. A language reference manual with a complete LALR (lookahead left recursive) grammar is provided in Appendix A.

CHAPTER 2
INTRODUCTION TO THE RESIDUE NUMBER SYSTEM

The following introduction and theoretical sections are derived from Mellott, *et al.* [38]. There exist a number of signal processing applications that demand high computational throughput in combination with high reliability, small size, and low power dissipation. In the past, high performance has come at the expense of reliability, size, power, and cost requirements. The prevalent arithmetic system used in digital hardware is two's complement. While two's complement is easy to use, it suffers from several impediments to achieving high performance. The speed of the adder in a two's complement system decreases at least with the logarithm of the word width of the adder due to the propagation of the carry term across the adder. The two's complement multiplier suffers not only from the "curse of carry," but also from quadratic growth of the required die area as the word width of the operands increases [39]. Multiplier structures continue to occupy large die area on modern VLSI microprocessors. Since the RNS is a carry-free arithmetic system, word widths of arbitrary size may be produced with no speed penalty in the adder. The size of the multiplier also grows linearly with respect to the word width of the multiplicands, rather than quadraticly as in two's complement schemes. The speed of RNS rithmetic elements, both addition and multiplication, is independent of the word width. Besides high performance, the RNS enables a high degree of fault-tolerance at the architectural level [40, 41]. Due to the high level of integration possible with RNS arithmetic elements, RNS is an enabling technology for ULSI (ultra large scale integration) systolic arrays [33, 42] and other high-order, integrated multi-processor architectures.

2.1 The Chinese Remainder Theorem

There are two large penalties in performing arithmetic in the two's complement system: the carry must propagate across the entire word for addition operations, and the size of the multiplier grows as the square of the width of the word. The Chinese Remainder Theorem (CRT) [43, 44] suggests a means of eliminating the carry propagation problem and of producing a multiplier that grows linearly with the width of the word. The RNS takes advantage of the isomorphism $\mathbb{Z}/M\mathbb{Z} \leftrightarrow \mathbb{Z}/p_1\mathbb{Z} \times \mathbb{Z}/p_2\mathbb{Z} \times \mathbb{Z}/p_3\mathbb{Z} \times \cdots \times \mathbb{Z}/p_L\mathbb{Z}$ given by the CRT. Throughout the remainder of this text, the notation $\mathbb{Z}_p$ (which is taken to be the ring $(\{0,1,2,\ldots,p-1\},\cdot,+)$) will be used to denote $\mathbb{Z}/p\mathbb{Z}$ since $\mathbb{Z}_p \cong \mathbb{Z}/p\mathbb{Z}$. The CRT is presented below.

Theorem 1 (The Chinese Remainder Theorem) *Let $M = \prod_{i=1}^{L} p_i$, where for $i,j \in \{1,2,3,\ldots,L\}$, $\gcd(p_i,p_j) = 1$ for all $i \neq j$, and each $p_i \in \mathbb{Z}^+$ (the positive integers). Then there exists an isomorphism ϕ: $\mathbb{Z}_M \leftrightarrow \mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \mathbb{Z}_{p_3} \times \cdots \times \mathbb{Z}_{p_L}$ described by the following.*

Let $m_i = M/p_i$, and $m_i m_i^{-1} \equiv 1 \pmod{p_i}$ for all $i \in \{1,2,3,\ldots,L\}$. If $X \in \mathbb{Z}_M$, let $\phi(X) = (x_1,x_2,x_3,\ldots,x_L)$ where $x_i \equiv X \pmod{p_i}$ for all $i \in \{1,2,3,\ldots,L\}$ then $X = \phi^{-1}(x_1,x_2,x_3,\ldots,x_L)$ is described by the following congruence

$$X \equiv \left\{ \sum_{i=1}^{L} m_i \langle m_i^{-1} x_i \rangle_{p_i} \right\} \pmod{M} \tag{2.1}$$

where $\langle \bullet \rangle_p$ indicates the unary (mod p) operation.

The CRT is the basis of the RNS. In the RNS, two's complement integers are converted to their L-tuple residue representation by the ring isomorphism ϕ: $\mathbb{Z}_M \leftrightarrow \mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \mathbb{Z}_{p_3} \times \cdots \times \mathbb{Z}_{p_L}$ described by the CRT. The numbers, which are in their L-tuple representation, may be added and multiplied component-wise and reconstructed via the CRT to form the correct result in $\mathbb{Z}_M$.

Generally, the moduli are chosen to be small enough that the multipliers may be implemented with the aid of reasonably small memory-based lookup tables. In a VLSI or ASIC implementation advanced memory technology could be leveraged.

2.2 Complex Residue Number System

The RNS may be used to perform computations with complex numbers by using RNS arithmetic elements to emulate the operations which would be performed using conventional arithmetic. The use of RNS arithmetic to perform complex operations is called complex RNS or CRNS. Take the Gaussian integers $a+jb, c+jd \in \mathbb{Z}_M[j]/\langle j^2+1\rangle$, and let ψ denote the isomorphism between the Gaussian integers and the CRNS:

$$\psi\colon\ \mathbb{Z}_M[j]/\langle j^2+1\rangle \leftrightarrow\ \mathbb{Z}_{p_1}\times\mathbb{Z}_{p_2}\times\mathbb{Z}_{p_3}\times\cdots\times\mathbb{Z}_{p_L}\times\mathbb{Z}_{p_1}\times\mathbb{Z}_{p_2}\times\mathbb{Z}_{p_3}\times\cdots\times\mathbb{Z}_{p_L}. \quad (2.2)$$

Then addition in the CRNS is performed as

$$\begin{aligned}(a+jb)+(c+jd) &= (a+c)+j(b+d) \\ &= \psi^{-1}\{\psi(a)+\psi(b)\}+ \\ &\quad j\psi^{-1}\{\psi(b)+\psi(d)\},\end{aligned} \quad (2.3)$$

and multiplication in the CRNS is performed as

$$\begin{aligned}(a+jb)\times(c+jd) &= (ac-bd)+j(ad+bc) \\ &= \psi^{-1}\{\psi(a)\psi(c)-\psi(b)\psi(d)\}+ \\ &\quad j\psi^{-1}\{\psi(a)\psi(d)+\psi(b)\psi(c)\}.\end{aligned} \quad (2.4)$$

While the complex addition takes only two additions, the complex multiplication takes four multiplications and two additions: the CRNS requires the same number of additions and multiplications as the Gaussian integers.

2.3 Quadratic Residue Number System

The quadratic RNS, or QRNS [41], is a variation upon the RNS which allows complex additions to be performed with two RNS additions and complex multiplications to be performed with two RNS multiplications. This enhancement is accomplished by encoding the real and imaginary components into two independent components. Given a prime p of the form $p = 4k + 1$ where $k \in \mathbb{Z}$, the congruence $x^2 \equiv -1 \pmod{p}$ has two solutions in the ring $(\mathbb{Z}_p, +, \cdot)$ that are multiplicative and additive inverses of one another. Let $\hat{j}$ and $\hat{j}^{-1}$ denote the two solutions to the above congruence. Define a mapping $\theta\colon\ \mathbb{Z}_p[j]/\langle j^2+1\rangle \rightarrow \mathbb{Z}_p \times \mathbb{Z}_p$ (where $\mathbb{Z}_p/\langle j^2+1\rangle$ is a sub-ring of $\mathbb{Z}_M/\langle j+1\rangle$) by

$$\begin{aligned} \theta(a+jb) &= (z, z^*) && (2.5)\\ z &\equiv (a+\hat{j}b) \pmod{p} && (2.6)\\ z^* &\equiv (a-\hat{j}b) \pmod{p}. && (2.7) \end{aligned}$$

The inverse mapping $\theta^{-1}\colon\ \mathbb{Z}_p \times \mathbb{Z}_p \rightarrow \mathbb{Z}_p[j]/\langle j^2+1\rangle$ is given by

$$\theta^{-1}(z, z^*) = \langle 2^{-1}(z+z^*)\rangle_p + j\langle 2^{-1}\hat{j}^{-1}(z-z^*)\rangle_p. \tag{2.8}$$

Suppose $(z, z^*), (w, w^*) \in \mathbb{Z}_p \times \mathbb{Z}_p$. Then the addition and multiplication operations in the ring $(\mathbb{Z}_p \times \mathbb{Z}_p, +, \cdot)$ are given by

$$(z, z^*) + (w, w^*) = (z+w, z^*+w^*), \tag{2.9}$$

and

$$(z, z^*)(w, w^*) = (zw, z^*w^*). \tag{2.10}$$

The isomorphic mappings θ and θ^{-1} are generally implemented via arithmetic elements and table lookup. Since the z and z^* channels are independent, parallel hardware may be constructed to perform operations on both channels at the same time without any communication between the channels. This parallelism allows a complex addition or multiplication to be performed in one cycle. While parallel hardware would allow a CRNS addition in one cycle, the multiplication in the CRNS requires two additions and four multiplications. Using the same amount of hardware as a QRNS multiplier-accumulator, a CRNS multiplier-accumulator would take twice as many cycles to complete a single multiply-accumulate operation.

2.4 Galois-Enhanced Quadratic Residue Number System

The QRNS requires a multiplier that takes N bit inputs and produces an N bit output. The multiplier could be implemented using either a direct implementation with modular correction or a lookup table. The primary disadvantage of these approaches is that despite the small size of the RNS adder, the multiplier is still large. By taking advantage of the properties of Galois fields [45], it is possible to simplify the implementation of an RNS multiplier.

For any prime modulus p there exists some $\alpha \in \mathbb{Z}_p$ that generates all non-zero elements of the field $GF(p)$. That is to say,

$$\{\alpha^i \mid i = 0, 1, 2, \ldots, p-2\} = GF(p) \setminus \{0\}. \tag{2.11}$$

Thus, all non-zero elements of $\mathbb{Z}_p$ may be uniquely represented by their number theoretic logarithms. These number theoretic logarithms may be added modulo $p-1$ to

produce multiplication,

$$\langle \alpha^{\langle i+j \rangle_{p-1}} \rangle_p = \langle \alpha^i \alpha^j \rangle_p. \tag{2.12}$$

Note that since zero is not an element of $GF(p) \setminus \{0\}$ the zero must be handled as an exception. Practically, this means that the inputs must be checked before the number theoretic logarithm to determine whether either one is a zero, and if one of the inputs is a zero, then the output of the multiplier should be set to zero.

The architecture of a Galois-enhanced QRNS, or GEQRNS, multiplier is illustrated in Figure 2.1 without the zero detection and handling indicated. The multiplier requires two duplicate 2^N-entry memories to perform the number theoretic logarithm, an adder to add the logarithms, and an 2^{N+1}-entry table to perform the modulo $p-1$ correction and number theoretic exponentiation. Note that while the modulo $p-1$ correction and number theoretic exponentiation represent two separate steps, they may be integrated into a single table. Alternatively, if a modular adder is used, the 2^{N+1}-entry table may be replaced with a 2^N-entry table. Typically, the multiplicands will be converted to the GEQRNS number theoretic logarithm form by the conversion engine which computes the residues of the integer inputs.

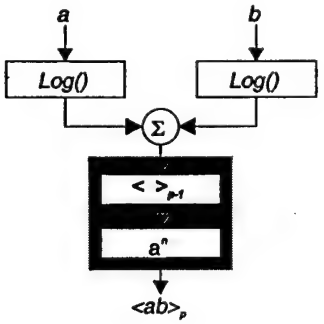


Figure 2.1: Block Diagram of a GEQRNS Multiplier

2.5 Logarithmic Residue Number System

The logarithmic RNS, or LRNS [45], is an enhancement to the GEQRNS whereby the results of addition operations are kept in the form of a number theoretic logarithm. Using the definition of p and α from Section 2.4, if $x, y \in GF(p) \setminus \{0\}$ then there exist unique $i, j \in \{0, 1, 2, \ldots, N-2\}$ such that $x = \alpha^i$ and $y = \alpha^j$. If x or y are zero then the arithmetic operation must be handled as an exception. Multiplication may be performed as in the GEQRNS using addition:

$$\langle xy \rangle_p = \langle \alpha^i \alpha^j \rangle_p = \langle \alpha^{\langle i+j \rangle_{p-1}} \rangle_p. \tag{2.13}$$

In the GEQRNS one would exponentiate a number theoretic logarithm before performing addition. The disadvantage of this is that two types of data are handled by the system and data conversions may need to be performed in some instances.

In the LRNS addition is performed in such a way as to keep the results in logarithmic form. Consider computing the sum $x + y$ in the LRNS:

$$\begin{aligned} \langle x + y \rangle_p &= \langle \alpha^i + \alpha^j \rangle_p \\ &= \langle \alpha^i (1 + \alpha^{\langle j-i \rangle_{p-1}}) \rangle_p. \end{aligned} \tag{2.14}$$

There exists a unique $k \in \{0, 1, 2, \ldots, N-2\}$ such that $\alpha^k = (1 + \alpha^{\langle j-i \rangle_{p-1}})$. The logarithm k is a function of the difference $\langle j - i \rangle_{p-1}$ (i.e., $k = f(\langle j - i \rangle_{p-1})$) and may be precomputed and stored in a table. Consequently, Equation 2.14 may be reduced to

$$\begin{aligned} \langle \alpha^i (1 + \alpha^{\langle j-i \rangle_{p-1}}) \rangle_p &= \langle \alpha^i \alpha^{f(\langle j-i \rangle_{p-1})} \rangle_p \\ &= \langle \alpha^{i + f(\langle j-i \rangle_{p-1})} \rangle_p. \end{aligned} \tag{2.15}$$

It is evident from this form that an LRNS addition operation can performed using one addition operation, one subtraction operation, and one small-table lookup operation. Since zero does not have a logarithm, if either or both of x and y are zero then the calculation must be handled as an exception. This is not difficult since zero is the additive identity. A block diagram of an LRNS multiplier-accumulator that takes LRNS operands as input and produces an LRNS result is shown in Figure 2.2. A value that is not a valid representation for the logarithm of a number in $GF(p) \setminus \{0\}$ is used to represent zero.

2.6 Previous Work in the RNS and Conclusions

In Mellott [37], a high performance multiprocessor architecture based upon the RNS is described. The Gauss machine [46, 38] is a hybrid systolic array and vector processor of GEQRNS processing elements which can achieve the peak equivalent of 320 million operations per second when performing complex arithmetic, see Figure 2.3. From this work it was determined that RNS systolic arrays are capable of performing many computations at rates limited only by the I/O capabilities of the processor. The I/O capabilities of the processor are ultimately limited by the VLSI technology: the practical limits on the number of I/O pads on a die represents a significant bottleneck. The issues involved in management of the number of pads versus the total die area are illustrated in Figure 2.4. The I/O problem is exacerbated by the limited speed of external connections versus internal connections. Furthermore as the pad count increases the minimum die area increases due to the requirement that the pads are arranged on the perimeter of a square or almost square die, see Figure 2.4(a). As I/O pads are added to a die, the area increases with the square of the number of pads. Improvements in process technology (i.e., scaling from an x μm to $x/2$ μm minimum feature size) do not provide any relief since pad size is determined by the physical

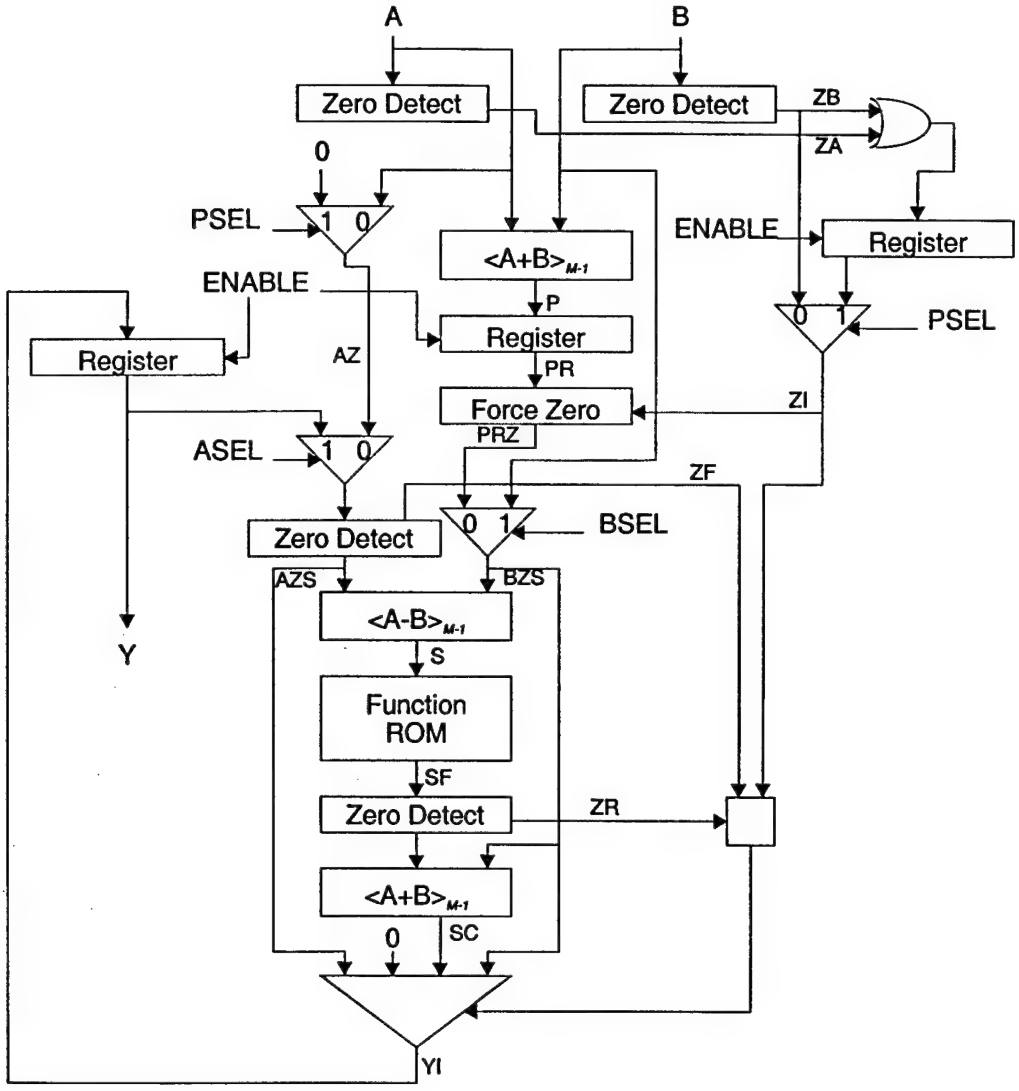


Figure 2.2: Block Diagram of an LRNS Multiplier-Accumulator

constraints of the external electrical connection, not the fabrication process. In fact, ongoing process and lithography improvements also serve to exacerbate the number of pads to die area ratio problem by increasing the amount of logic that can be placed on-chip without improving the number of pads, see Figure 2.4(b). Ultimately there is no way to add enough pads to supply data to a large scale processor that uses all of the die area for arithmetic elements.

The conclusion that follows from this analysis is that data must be loaded on-chip and substantial processing must be done on that data to have any hope of achieving

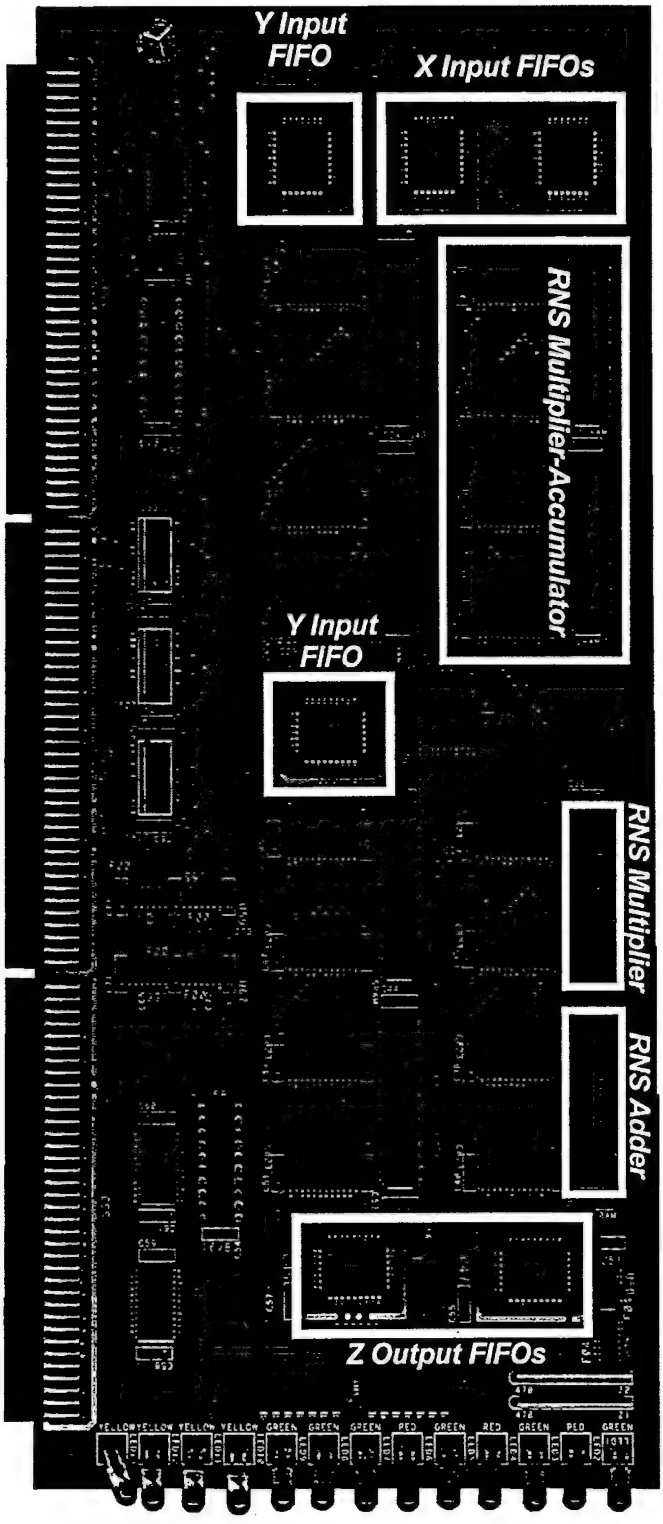


Figure 2.3: Photograph of Gauss Machine Single Channel, Quad Processor Card

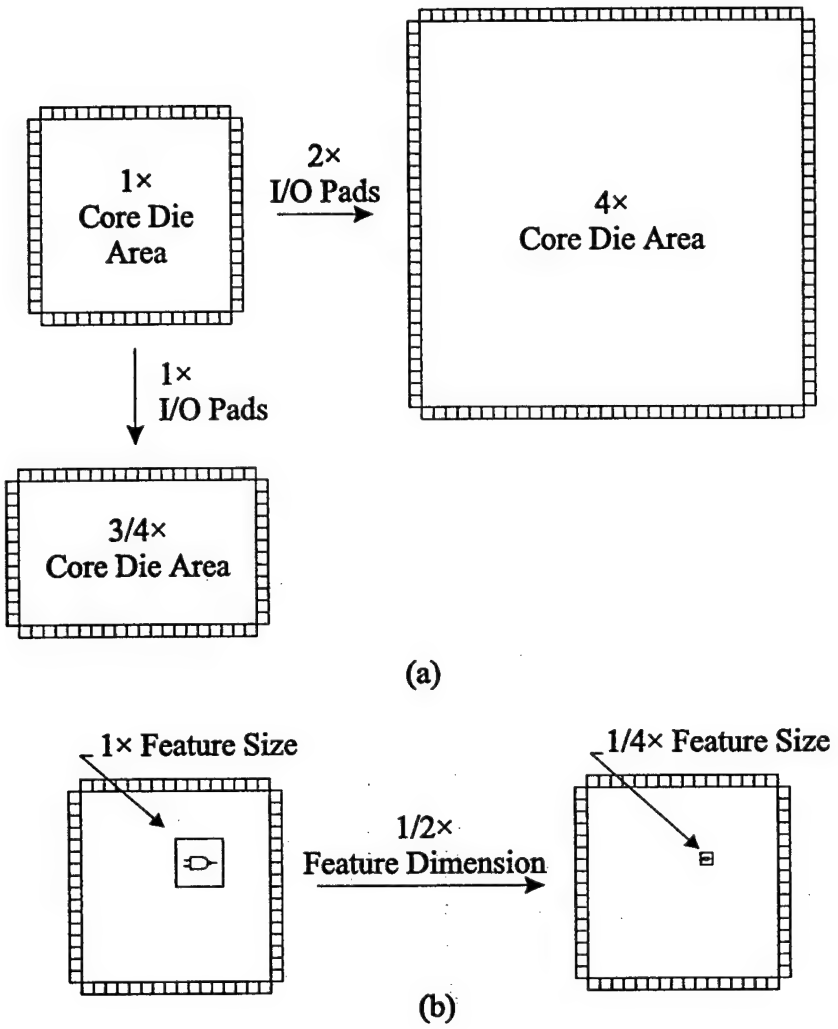


Figure 2.4: Illustration of (a) Pad Quantity to Area Ratio Management Options and (b) Impact of Process Improvements on Pad Quantity to Area Ratio

optimal use of large arrays of RNS processors. To enable some sort of optimally utilized RNS signal processor to be used in general purpose signal processing applications, a highly integrated RNS processor must be designed. A VLSI implementation must include the following features:

- an RNS processor,
- RNS/integer forward/reverse conversion hardware,
- a conventional arithmetic processor,

- *substantial* on-chip memory to alleviate data I/O bottlenecks, and
- an independent data I/O capability to shuttle data between on-chip and off-chip memories.

From these specifications, the most obvious opportunities to accelerate DSP operations using the RNS are either to loosely couple an RNS accelerator to a microprocessor (DSP or general purpose), or to tightly couple RNS architectural elements to a DSP microprocessor. Since the latter approach implies multiple function units within a single instruction set architecture, the architecture should support multiple issue.

CHAPTER 3
THE ATHENA SENSOR ARITHMETIC PROCESSOR

The Athena Sensor Arithmetic Processor[1] (ASAP) has been important in motivating this study. The ASAP device provided motivation to examine the integration of VLIW architectural techniques with digital signal processors, and in particular, the integration of the ASAP processor technology in the VLIW environment. To date, RNS processor implementations have been hardwired to specific applications. To alleviate the burden of custom engineering of hardware solutions that use the RNS, it is necessary to integrate an RNS technology into an environment where applications can be developed at the software level. A VLIW approach was selected since it offers the means to manage multiple functional units, as would be needed in a general purpose digital signal processor that uses an RNS processor technology.

The primary goal of the Athena Sensor Arithmetic Processor (ASAP) device is to perform video rate DFTs using the Good-Thomas FFT [9, 7, 11] algorithm. To support the target 231×231 frame size, it is necessary to support Rader prime DFTs [10, 7, 12] of length three, seven, and eleven. Therefore convolutions of length two, six, and ten must be supported. Since two dimensional DFTs can be constructed using one dimensional DFTs, it is reasonable to expect the device to perform a one dimensional DFT using on-chip resources. To this end at least three banks of 256 words of on-chip SRAM (static RAM) are desirable. For other applications larger on-chip memories may be desirable; however, for the ASAP design this RAM size

[1]The Athena Sensor Arithmetic Processor was developed by the author as a consultant to The Athena Group, Inc. in support of U.S. Air Force contract F08630-93-0072.

and type was the most logical. One of the three banks can supply the z or z^* components (assuming QRNS coding) for the DFT while another bank can contain the coefficients for the DFT, and the last bank can be used to accumulate the results. Since the designed LRNS arithmetic elements have an extremely voracious appetite for data, inclusion of a fourth bank of memory to allow data I/O to be performed concurrently with computations is warranted.

Since data locality can be insured, the data RAMs are connected to the processor and external data I/O path via a configurable switch. The configuration is written so as to select individual memories as operand sources, results storage, or to connect a memory block to the external data I/O path. A block diagram of the resulting architecture is shown in Figure 3.1.

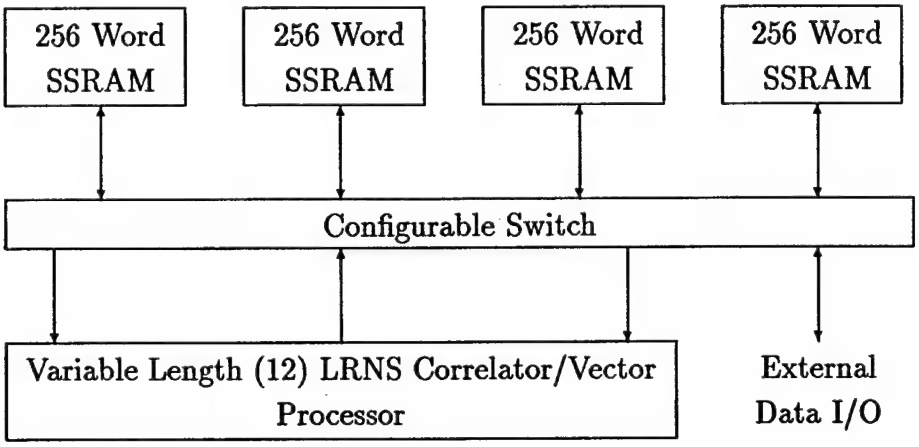


Figure 3.1: Block Diagram of ASAP Architecture

The ASAP chip is a four moduli (241, 233, 229, 197) SIMD (single-instruction, multiple-data) processor. There are twelve LRNS arithmetic elements configured as a variable length linear correlator/convolver. Circular convolution is achieved by restarting the convolution operation. The processor array may also be used for vector addition, multiplication, and multiply-accumulate operations. To ensure adequate data bandwidth to support computation, there are four 256 word synchronous static

RAMs (SSRAMs) that are used for processor data. Provisions are made for parallelism by allowing RAM I/O concurrent with computations for those RAM blocks not involved in the current computation, and by allowing arithmetic and convolution operations concurrent with recovery of previous results from the convolver. Control of this first generation of large scale devices is provided entirely by external inputs to the device for maximum flexibility.

The ASAP processor is fabricated in the MOSIS (metal-oxide semiconductor implementation service) 0.8 μm triple-metal CMOS (complementary metal oxide semiconductor) process (Hewlett-Packard) and packaged in a 108 pin ceramic pin grid array package. An annotated die photo of the device is shown in Figure 3.2. There are four processor "quadrants," and each quadrant is independent except for control, clocking, and power, which are shared among all four quadrants. Within each processor quadrant the four memories are clearly visible, as are the twelve LRNS multiplier-accumulators that comprise the array processor. Total die area is 38.6 mm^2. The core area (die size minus pads) is 31.8 mm^2. The forty-eight eight bit LRNS processors plus control and data buses that form the thirty-two bit length twelve convolver/correlator occupy only 19.6 mm^2 of the die area. Each individual LRNS multiplier-accumulator core occupies only 0.246 mm^2 (210 μm $\times$ 1170 μm) of die area. The design scales directly into the 0.6 μm MOSIS (Hewlett-Packard) process that went online in Fall of 1995 — the above *core* areas may be multiplied by 0.5625 to arrive at the core die areas in the 0.6 μm process.

3.1 Test Chip

A small test chip was fabricated before the large ASAP device was fabricated to test the function and performance of the key constituent cells of the ASAP device. The device implements a single GEQRNS multiplier-accumulator unit. The test device

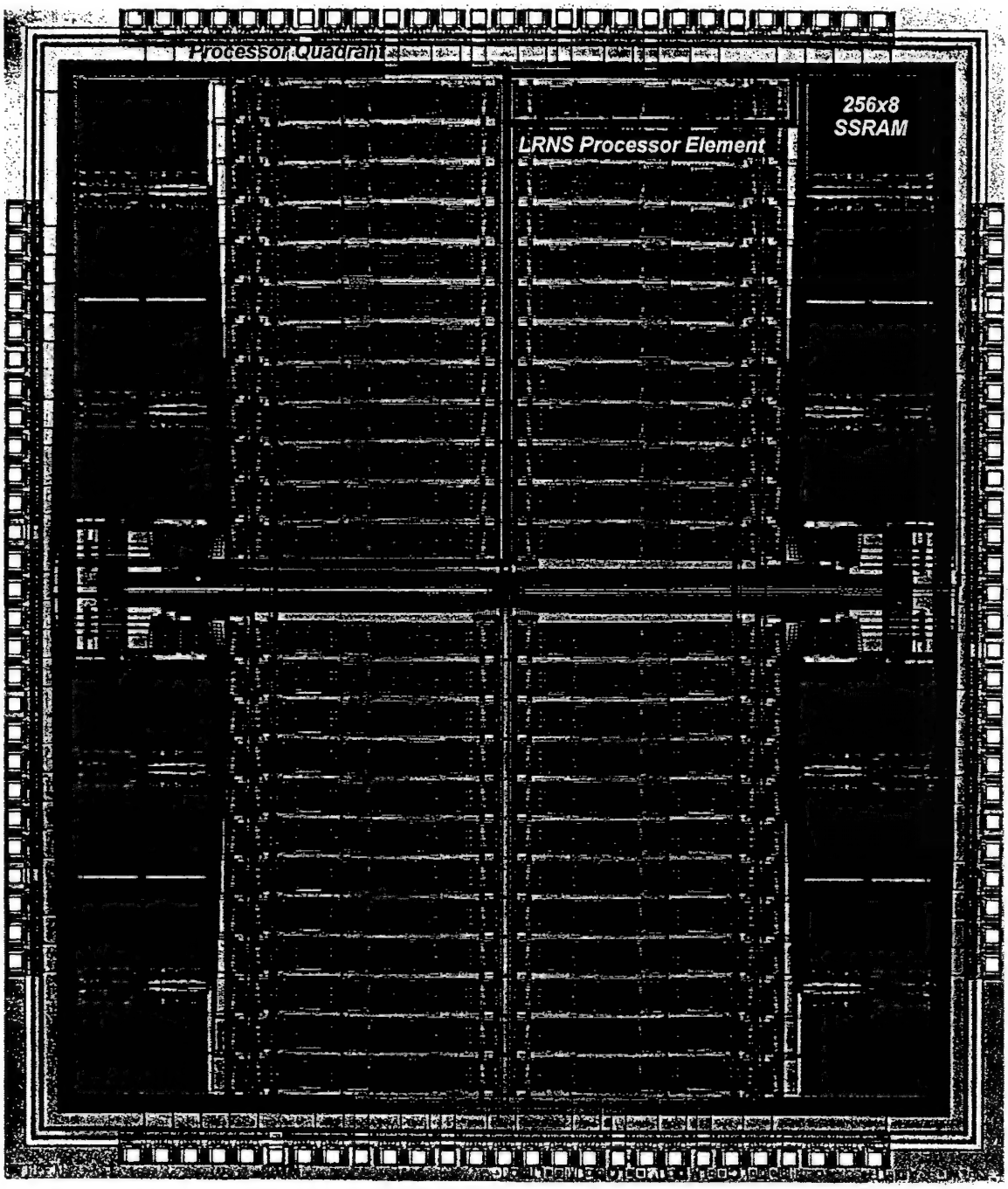


Figure 3.2: Annotated Die Photograph of the ASAP Device

was fabricated using the standard TinyChip frame under the MOSIS 2.0 μm CMOS process. In addition to the basic cells required by the ASAP device, the test device also included enhanced observability features that could not have been reasonably provided for on the full-scale ASAP device due to packaging constraints.

The test device is packaged in a forty pin ceramic DIP (dual in-line package) with a pinout as given in Figure 3.3 and signals as described in Table 3.1. A block diagram of the arithmetic unit included on the device is shown in Figure 3.5. The device has two data inputs, the A bus and B bus. There is a single data output, the Y bus. There are six digital control inputs, the analog threshold input for the ROM (read-only memory) sense amplifiers, and a clock signal that is buffered and drives the register elements. There are two positive rail (VCC) and ground (GND) inputs for power, one of each used to drive the I/O ring and core logic.

The test device is shown in a test fixture with the cavity exposed in Figure 3.4. Due to the packaging constraints of the TinyChip format a great deal of die area is wasted in this implementation. Using the same die area as the TinyChip, but with a slightly modified geometry, two multiplier-accumulators could have been placed on the device.

Using undedicated pins, two test structures were added to the test device. First, a single true single phase positive-edge triggered enable D register was added. This register was included because of it was an untested design and its functionality is dependent upon dynamic circuit performance. The register's input is the A7 data operand input, its enable signal is the A6 data operand input, and its buffered output is presented on the dedicated pin A7OUT.

The second test structure was an eight-to-one MUX with its inputs connected directly to the eight ROM sense amplifier outputs. Like the register element, the sense amplifier represents one of the riskier portions of the design due to its analog

Table 3.1: ASAP Test Chip Pin Descriptions

Signal Name	Input/Output	Pin Numbers	Description
A0–A7	I	37, 38, 39, 40, 1, 4, 3, 2	A operand input.
B0–B7	I	34, 33, 32, 31, 29, 28, 27, 26	B operand input.
Y7–Y0	O	6, 7, 8, 9, 11, 12, 13, 14	Y result output.
ASEL	I	19	Adder A operand mux select. One selects the output of the P mux while zero selects the Y bus.
BSEL	I	17	Adder B operand mux select. One selects the B bus while zero selects the P bus.
PSEL	I	18	P mux select. One forces zero output while zero passes the A bus through.
FSEL	I	20	Feedback bus mux select. One forces zero into the accumulator/output register while zero passes the F bus through.
AENABLE	I	21	Adder register enable.
MENABLE	I	23	Multiplier register enable.
THRESH	Analog In	24	ROM sense amplifier threshold. Never tie lower than 2V.
PHI	I	36	Clock input.
VCC	I	30	Logic power supply.
VCC	I	5, 15	I/O power supply.
GND	I	10	Logic ground.
GND	I	25, 35	I/O ground.
A7OUT	O	16	Registered copy of A7. Controlled by A6.
EN	I	3	Register enable for A7 output register. Same pin as A6.
TA0-TA2	I	26, 27, 28	ROM test output mux select. Same pins as B5-B7.
TOUT	O	22	ROM test mux output.

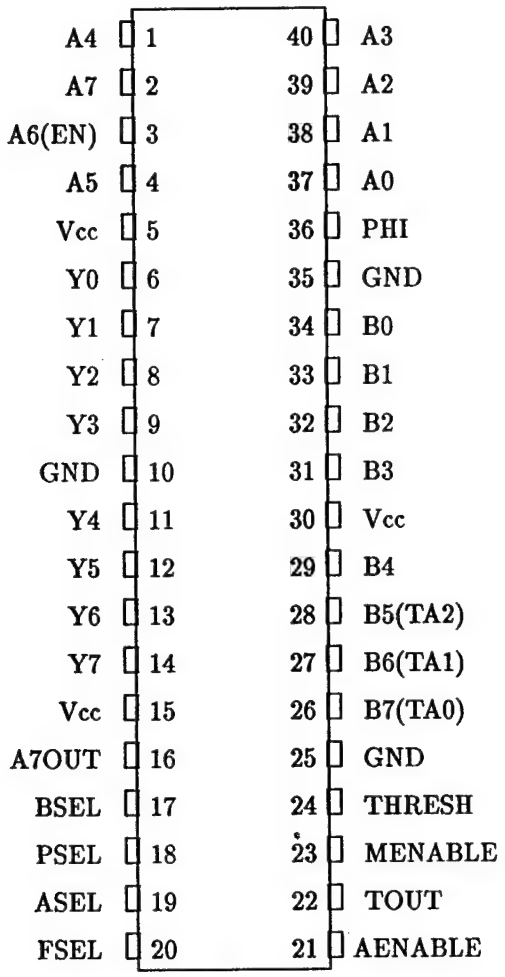


Figure 3.3: Pinout of the Test Chip

nature. The select inputs for the MUX, TA0, TA1, and TA2, are also B5, B6, and B7, respectively. This is an acceptable re-use of these inputs since the processor can be halted by negating the AENABLE and MENABLE signals. The output of the MUX is sent to the dedicated output TOUT. By cycling TA0–TA2 and monitoring TOUT the output of the ROM can be monitored. This allowed for selection of the THRESH analog input to the ROM sense amplifiers during testing.

Extensive testing of the device determined that the device performed as expected. Testing included complete coverage of each arithmetic and memory function. Com-

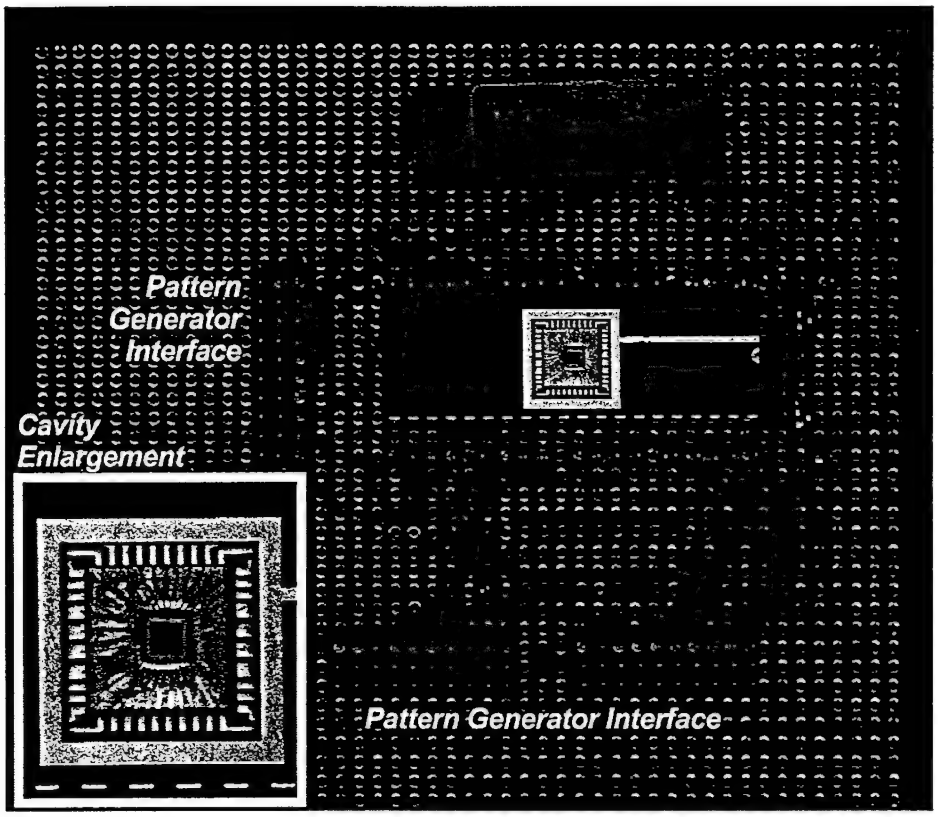


Figure 3.4: ASAP Test Chip in Test Fixture

plete coverage of the test vectors was aided by the design which allowed each logic element to be tested in isolation.

3.2 Detailed Architecture Description

3.2.1 Synchronous static RAM

The synchronous SRAM consists of a 256×8 static RAM, an address input register, data input register, data output register, and command input register. A block diagram of the memory is shown in Figure 3.6. The registers are clocked by the system clock and are enabled by RAMEN. The write enable (WE) signal is active high while the read enable (RE*) signal is active low. The WE signal must be asserted and clocked into the command register in order for a write to execute. Likewise, the

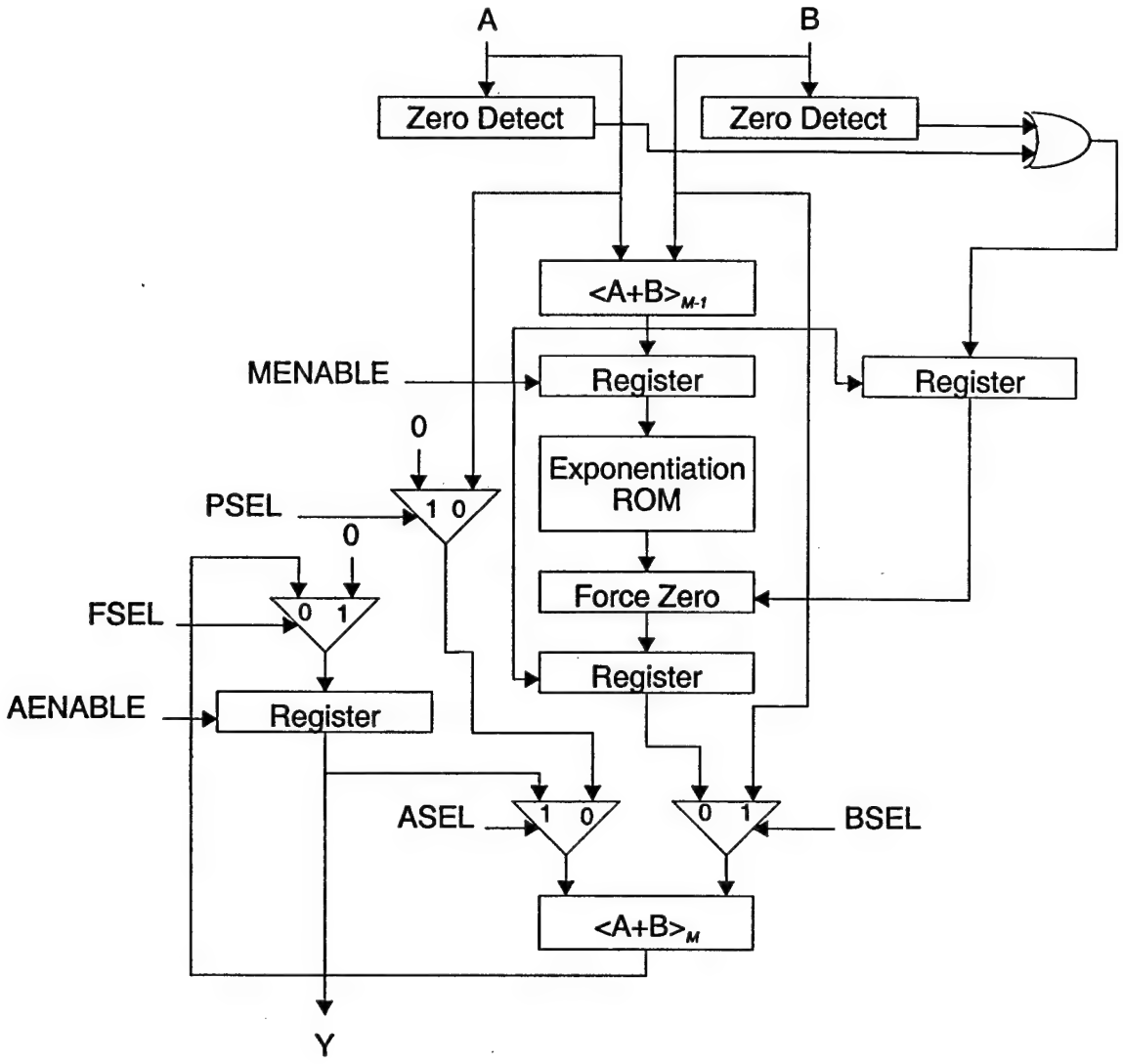


Figure 3.5: Block Diagram of Modular Multiplier/Adder/Accumulator Arithmetic Element

RE* signal must be asserted and clocked into the command register in order for a read to execute.

The operation of the synchronous SRAM is summarized in Table 3.2. Note that the pipeline is essentially two levels deep for both reads and writes. For example, in a write operation an address, data, and a write command are presented to the RAM's inputs. On the first rising clock edge the address, data input and command are clocked into the registers. The write is not complete such that the data is available

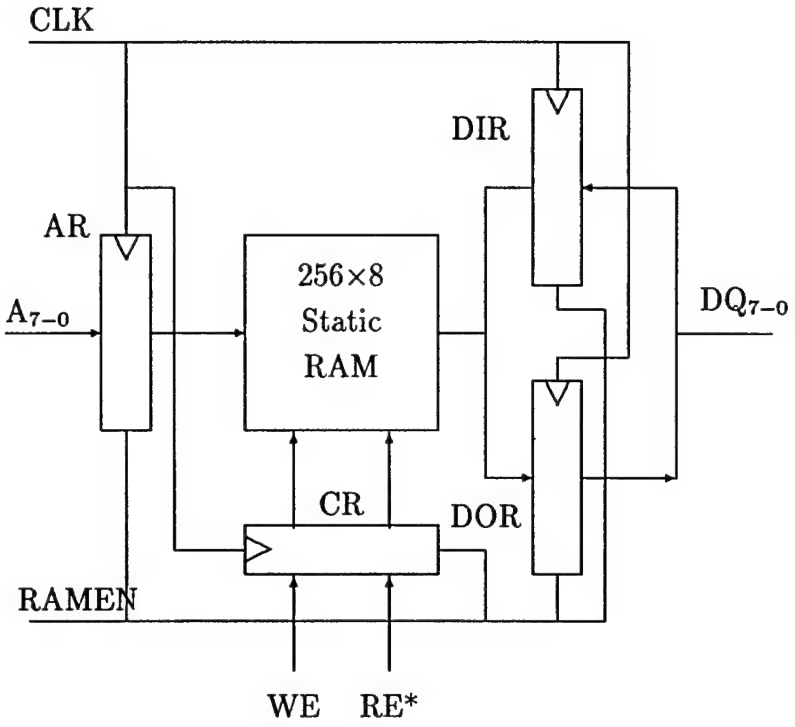


Figure 3.6: Block Diagram of Synchronous SRAM

Table 3.2: Synchronous SRAM Command Effects

CLK	RAMEN	WE	RE*	Effect
X	X	L	X	DOR → DQ_{7-0}
X	L	H	X	—
↑	H	L	X	A_{7-0}– → AR DQ_{7-0}– → DIR SRAM(AR–) → DOR
↑	H	H	H	A_{7-0}– → AR DQ_{7-0}– → DIR DIR- → DOR DIR+ → SRAM(AR+)
↑	H	H	L	A_{7-0}– → AR DQ_{7-0}– → DIR SRAM(AR–) → DOR

The +/– indicate signal status after/before the clock edge.

for reading until the next clock edge. Likewise, for a read operation, the address and read command are presented before the first clock edge. The data is clocked into

the data output register (DOR) on the next (second) clock edge, after which it is externally available.

3.2.2 Data switch

Connections between processor inputs and outputs, memories, and the external data I/O bus are performed by the data switch array. This array consists of four eight bit wide four-to-one MUXes connecting the four RAM blocks with the A and B processor bus inputs, Y processor shift register output, and the external data I/O bus. The configuration of this switch is controlled by the elements of the command and configuration register. A block diagram of the data switch is given in Figure 3.7.

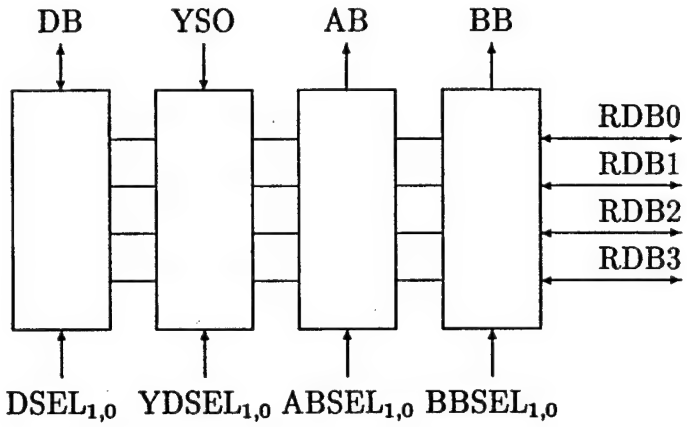


Figure 3.7: Data Switch Block Diagram

3.2.3 Command and configuration register

The operation of the ASAP device is controlled by an internal configuration register. This command register is a thirty-two bit write-only register that is connected to the data bus I/O lines. Write is enabled to this register by asserting the CMDREN signal. The command register controls the configuration of the RAM interconnection and the connection of the individual processor elements to the 'A' and 'B' shift registers.

Table 3.3: Command Register Map

Signal	Register	Signal	Register	Signal	Register	Signal	Register
DA_7	BDSEL0	DA_6	ADSEL0	DA_5	BDSEL1	DA_4	ADSEL1
DA_3	BDSEL2	DA_2	ADSEL2	DA_1	BDSEL3	DA_0	ADSEL3
DB_7	BDSEL4	DB_6	ADSEL4	DB_5	BDSEL5	DB_4	ADSEL5
DB_3	BDSEL6	DB_2	ADSEL6	DB_1	BDSEL7	DB_0	ADSEL7
DC_7	YDSEL1	DC_6	YDSEL0	DC_5	DSEL1	DC_4	DSEL0
DC_3	ABSEL0	DC_2	ABSEL1	DC_1	BBSEL0	DC_0	BBSEL1
DD_7	BDSEL8	DD_6	ADSEL8	DD_5	BDSEL9	DD_4	ADSEL9
DD_3	BDSEL10	DD_2	ADSEL10	DD_1	BDSEL11	DD_0	ADSEL11

The A and B shift registers are controlled by the $ADSEL_{11-0}$ and $BDSEL_{11-0}$ elements of the command register. A one in the register causes that element of the A or B shift register to take input from the A or B bus (respectively) while a zero causes that element to take an input from the previous element of the shift register.

The $ABSEL_{1-0}$, $BBSEL_{1-0}$, $DSEL_{1-0}$, and $YDSEL_{1-0}$ signals control the operation of the switch between memory banks, the processors, and the external data I/O interface. These selects should not be placed into contention, although the most egregious contentions are precluded by design.

3.2.4 LRNS correlator processor

Twelve LRNS arithmetic elements are arranged with input operands that come from shift registers that shift in the opposite direction, thus allowing correlation and convolution operations to be executed. Results are shifted out of the arithmetic elements using another shift register. This architecture is detailed in Figure 3.8.

Each register in the input shift registers can take inputs either from the preceding register in the shift register or from an external bus. This arrangement allows the processor to easily be configured as a variable length correlator, thus reducing the pipeline start delays associated with short length correlations such as those used in the Rader prime DFTs that are components of the Good-Thomas DFT. The registers

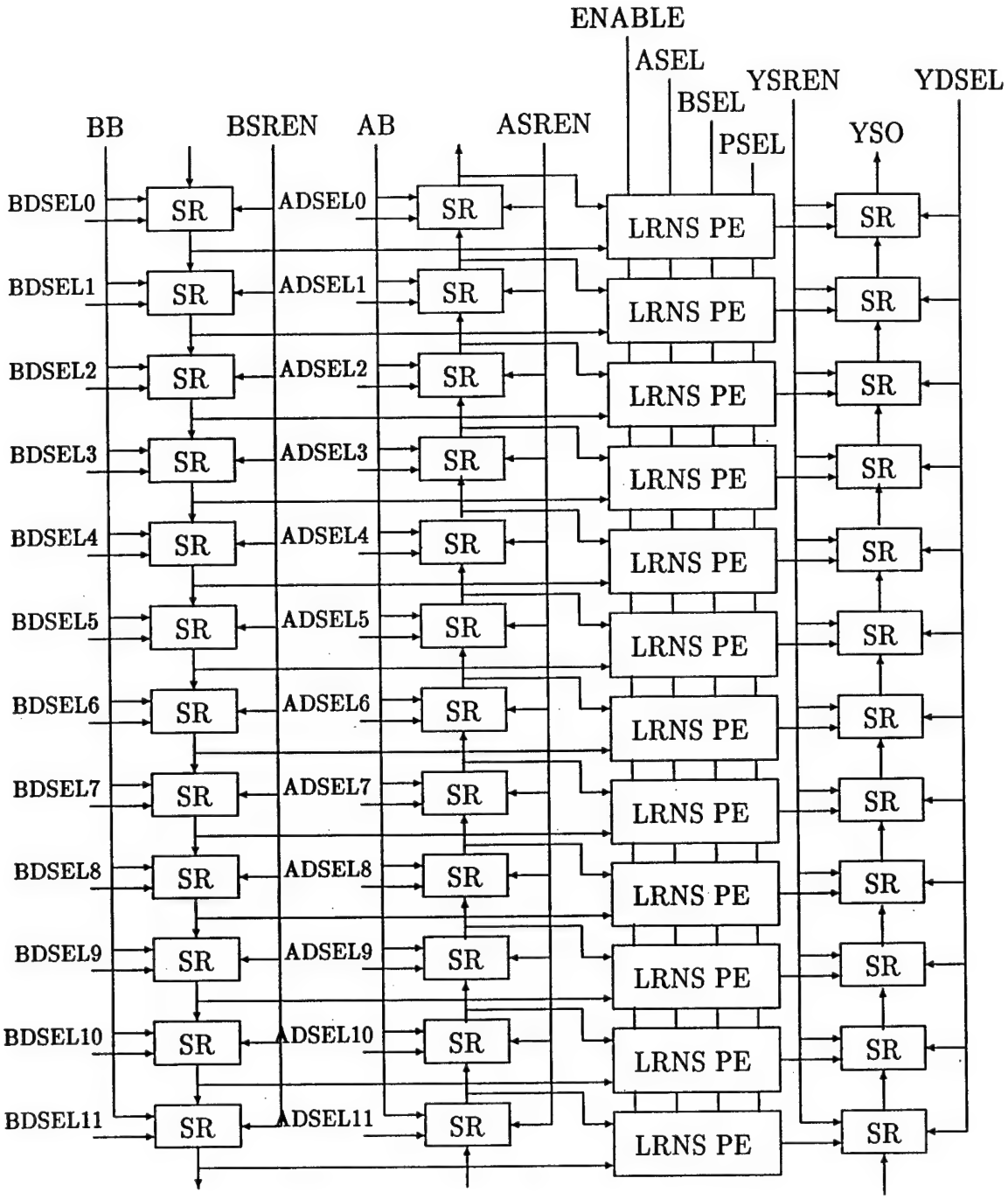


Figure 3.8: LRNS Correlator Processor

in the output shift register either take their inputs from the LRNS processor elements in parallel or from the previous register in the shift register chain. The data I/O and control signals for the correlator are given in Table 3.4.

Table 3.4: Correlator Data I/O and Control Signals

Signal	Description
AB_{7-0}	A shift register input bus.
BB_{7-0}	B shift register input bus.
YSO_{7-0}	Y shift register output.
$ADSEL_{11-0}$	A shift register data source select. One selects the AB bus while zero selects the previous shift register. These signals come from the command and configuration register.
$BDSEL_{11-0}$	B shift register data source select. Operates like ADSEL.
YDSEL	Y shift register data source select. One selects the LRNS processor element output while zero selects the previous shift register.
ASREN	A shift register enable.
BSREN	B shift register enable.
YSREN	Y shift register enable.

3.2.5 LRNS processor element

A detailed block diagram of the implemented LRNS arithmetic unit is depicted in Figure 2.2. A simplified version of the same block diagram is shown in Figure 3.9. The simplified version of the diagram is adequate to describe the functional operation of the LRNS processor element to the user of the ASAP device.

The processor element is controlled by three select signals and one enable signal that enables the pipeline. The use of the three select signals is summarized in Table 3.5. Referring to Figure 3.9, it would appear that certain combinations of select inputs might be useful but are marked as invalid operations in Table 3.5. To understand why these are invalid operations one must turn to the detailed block diagram of the LRNS processor element given in Figure 2.2.

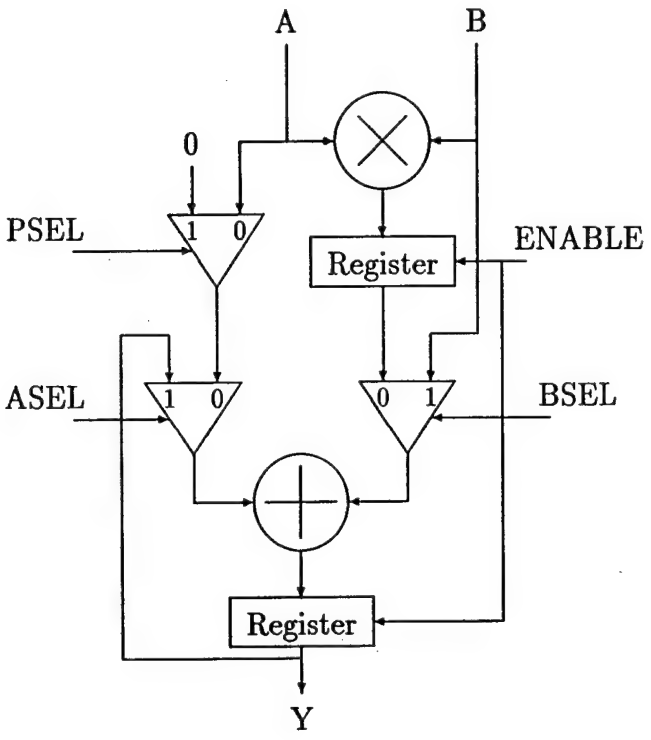


Figure 3.9: Simplified Block Diagram of Modular Multiplier/Adder/Accumulator Arithmetic Element

Table 3.5: LRNS Control Signals and Operations

PSEL	ASEL	BSEL	Operation
0	X	0	Invalid Operation
0	0	1	Vector Additions $(A + B \rightarrow Y)$
0	1	1	Vector Accumulate $(B + Y \rightarrow Y)$
1	0	0	Vector Multiply $(AB \rightarrow Y)$
1	X	1	Invalid Operation
1	1	0	Multiply Accumulate $(AB + Y \rightarrow Y)$

Depending upon the operation being performed, the length of the pipeline is either one or two registers: when the multiplier is used the length is two registers, and when the multiplier is not used the length of the pipeline is one register. In some circumstances an extra cycle might need to be inserted before switching from one operation type to another. For example, to switch from vector multiplication to vector addition an extra cycle between the last data for the vector multiplication

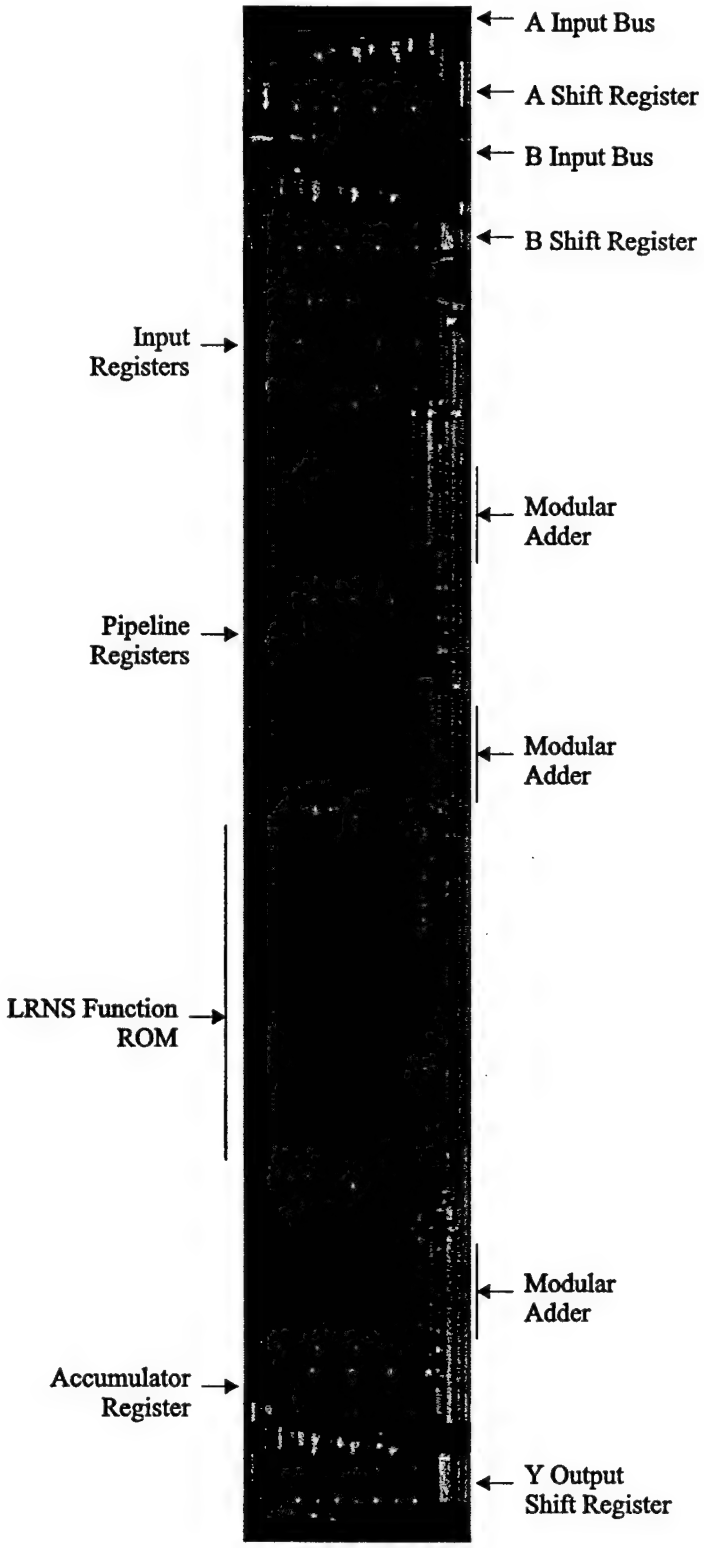


Figure 3.10: Annotated Die Photograph of LRNS Processor Element

must be executed so as to allow the final vector multiplication result to propagate through the pipeline.

The LRNS processor element does not have an explicit "reset" signal. Instead the processor must be reset programatically. Whether a reset operation is required will depend upon the operation being performed: vector addition and multiplication do not require initialization of the processor; while vector accumulation and multiply-accumulate do require initialization. Initialization is accomplished by setting the data input and control signals according to Table 3.6. The signals listed must be asserted for two clock cycles so that the initialization can propagate through the pipeline. Computation can begin immediately after the initialization.

Table 3.6: LRNS Processor Initialization Inputs

A	0_{LRNS}
B	0_{LRNS}
ASEL	0
BSEL	0
PSEL	1
ENABLE	1

3.3 Execution of Basic Algorithms

This section describes the execution of some basic algorithms on the ASAP correlation processor. The algorithms shown are processor initialization, vector addition, vector accumulation, vector multiplication, vector multiply-accumulation, and convolution. These operations are the algorithmic building blocks of many DSP applications.

3.3.1 Initialization

The exact command sequence for processor initialization to a reset state is given in Table 3.7. Note that the values given for AB and BB in the table are actual encoded LRNS values (FF_{16} is an encoded LRNS zero), not hexadecimal equivalents.

Table 3.7: Processor Initialization Sequence

N	AB	ABS	ASREN	BB	BBS	BSREN	ENABLE	ASEL	BSEL	PSEL	YDS	YSREN	YSO
0	FF	FFF	1	FF	FFF	1	X	X	X	X	X	X	U
1	XX	XXX	0	XX	XXX	0	1	0	0	1	X	X	U
2	XX	XXX	0	XX	XXX	0	1	0	0	1	X	X	U

3.3.2 Basic vector operations

The vector operations are characterized by using only one processor in the processor chain. The vector multiplication and vector addition operations do not require that the processor be initialized while the vector accumulate operation and multiply-accumulate operation both require that the processor be initialized before the computation begins.

Vector multiplication of length $N+1$ vectors $\mathbf{a}$ and $\mathbf{b}$ to produce the length $N+1$ vector $\mathbf{y}$ is given as $y_i = a_i b_i$ for all $i \in \{0, 1, 2, \ldots, N\}$. The command sequence for a vector multiplication is illustrated in Table 3.8. The total pipeline delay exhibited in this operation is four cycles: one due to the input register, two due to the LRNS processor element in vector multiplication configuration, and one due to the output shift register. The pipeline operation of a vector multiplication is illustrated in Figure 3.11.

Table 3.8: Vector Multiplication Procedure

N	AB	ABS	ASREN	BB	BBS	BSREN	ENABLE	ASEL	BSEL	PSEL	YDS	YSREN	YSO
0	a_0	001	1	b_0	001	1	X	X	X	X	X	X	U
1	a_1	001	1	b_1	001	1	1	0	0	1	X	X	U
2	a_2	001	1	b_2	001	1	1	0	0	1	X	X	U
3	a_3	001	1	b_3	001	1	1	0	0	1	1	1	U
4	a_4	001	1	b_4	001	1	1	0	0	1	1	1	y_0
5	a_5	001	1	b_5	001	1	1	0	0	1	1	1	y_1
6	...	"	"	...	"	"	"	"	"	"	"	"	...
7	a_N	001	1	b_N	001	1	1	0	0	1	1	1	y_{N-4}
8	XX	XXX	X	XX	XXX	X	1	0	0	1	1	1	y_{N-3}
9	XX	XXX	X	XX	XXX	X	1	0	0	1	1	1	y_{N-2}
10	XX	XXX	X	XX	XXX	X	X	X	X	X	1	1	y_{N-1}
11	XX	XXX	X	XX	XXX	X	X	X	X	X	X	X	y_N

Vector addition of length $N+1$ vectors $\mathbf{a}$ and $\mathbf{b}$ to produce the length $N+1$ vector $\mathbf{y}$ is given as $y_i = a_i + b_i$ for all $i \in \{0, 1, 2, \ldots, N\}$. The command sequence for a vector addition is illustrated in Table 3.9. The total pipeline delay exhibited in this operation is three cycles: one due to the input register, one due to the LRNS

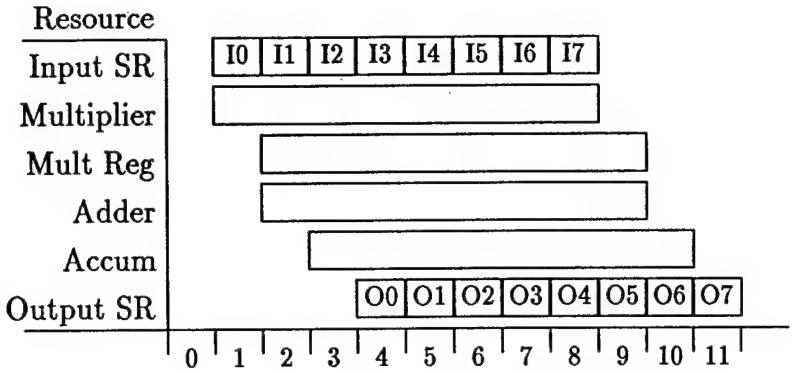


Figure 3.11: Pipeline Operation for Vector Multiplication

processor element in vector addition configuration, and one due to the output shift register. Pipeline operation of vector addition is illustrated in Figure 3.12.

Table 3.9: Vector Addition Procedure

N	AB	ABS	ASREN	BB	BBS	BSREN	ENABLE	ASEL	BSEL	PSEL	YDS	YSREN	YSO
0	a_0	001	1	b_0	001	1	X	X	X	X	X	X	U
1	a_1	001	1	b_1	001	1	1	0	1	0	X	X	U
2	a_2	001	1	b_2	001	1	1	0	1	0	1	1	U
3	a_3	001	1	b_3	001	1	1	0	1	0	1	1	y_0
4	a_4	001	1	b_4	001	1	1	0	1	0	1	1	y_1
5	a_5	001	1	b_5	001	1	1	0	1	0	1	1	y_2
6	...	"	"	...	"	"	"	"	"	"	"	"	...
7	a_N	001	1	b_N	001	1	1	0	1	0	1	1	y_{N-3}
8	XX	XXX	X	XX	XXX	X	1	0	1	0	1	1	y_{N-2}
9	XX	XXX	X	XX	XXX	X	1	0	1	0	1	1	y_{N-1}
10	XX	XXX	X	XX	XXX	X	X	X	X	X	X	X	y_N

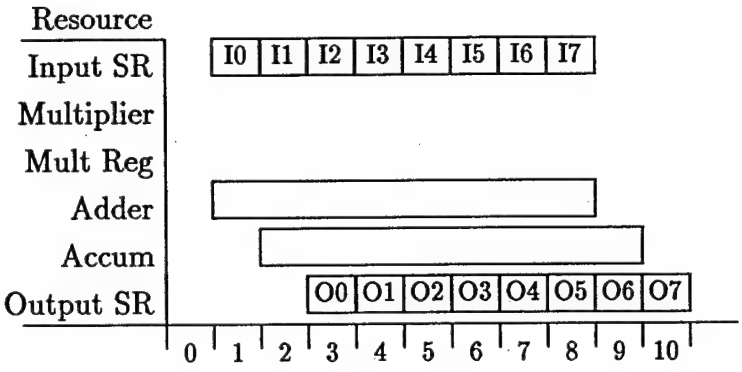


Figure 3.12: Pipeline Operation of Vector Addition

The vector accumulate and multiply-accumulate operations require that the processor element be initialized to zero. The procedure to accumulate an $N+1$ element vector b is given as $y = \sum_{i=0}^{N} b_i$ and is illustrated in Table 3.10. The initialization

of the accumulator spans steps zero through two, although the dataflow may start at step two. The total pipeline delay incurred in this operation is three cycles: one due to the input shift register, one due to the LRNS processor element, and one due to the output shift register. Note that the data input must be presented to the B broadcast bus, BB. Also note that the Y shift register is only programmed to sample the final result so the Y shift register is not committed until the final step of the computation. Consequently data from a previous operation may be shifted out of the processor while an accumulate operation is underway. Alternatively, the Y shift register can sample the LRNS processor element's Y output on each cycle allowing intermediate results to be monitored on YSO. An example of the pipeline's operation is given in Figure 3.13.

Table 3.10: Vector Accumulate Procedure

N	AB	ABS	ASREN	BB	BBS	BSREN	ENABLE	ASEL	BSEL	PSEL	YDS	YSREN	YSO
0	FF	001	1	FF	001	1	X	X	X	X	X	X	U
1	XX	XXX	0	XX	XXX	0	1	0	0	1	X	X	U
2	XX	XXX	0	b_0	001	1	1	0	0	1	X	X	U
3	XX	XXX	X	b_1	001	1	1	1	1	0	X	X	U
4	XX	XXX	X	b_2	001	1	1	1	1	0	X	X	U
5	"	"	"	...	"	"	"	"	"	"	"	"	"
6	XX	XXX	X	b_N	001	1	1	1	1	0	X	X	U
7	XX	XXX	X	XX	XXX	X	1	1	1	0	X	X	U
8	XX	XXX	X	XX	XXX	X	X	X	X	X	1	1	U
9	XX	XXX	X	XX	XXX	X	X	X	X	X	X	X	y

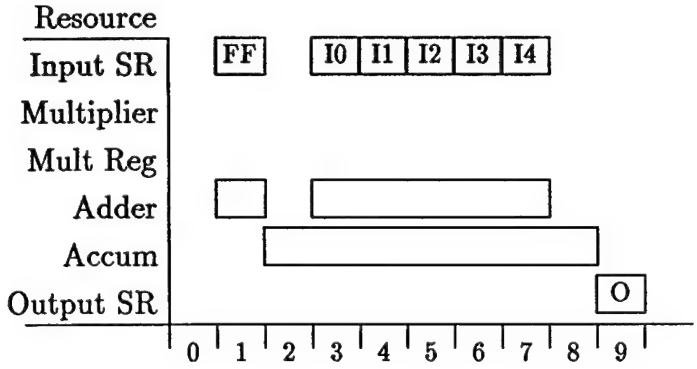


Figure 3.13: Pipeline Operation of Vector Accumulate

The vector multiply-accumulate procedure is very similar to the vector accumulate procedure described above. A procedure to multiply-accumulate two length $N+1$

vectors **a** and **b** to produce a scalar result $y = \sum_{i=0}^{N} a_i b_i$ is given in Table 3.11. The total pipeline delay incurred in this operation is four cycles: one cycle for the input operand shift registers, two cycles for the LRNS processor element, and one cycle for the Y shift register. The comments about the Y shift register in the vector accumulate operation also apply for the vector multiply-accumulate operation. An example of the pipeline operation of a multiply-accumulate operation is given in Figure 3.14.

Table 3.11: Vector Multiply-Accumulate Procedure

N	AB	ABS	ASREN	BB	BBS	BSREN	ENABLE	ASEL	BSEL	PSEL	YDS	YSREN	YSO
0	FF	001	1	FF	001	1	X	X	X	X	X	X	U
1	XX	XXX	0	XX	XXX	0	1	0	0	1	X	X	U
2	a_0	001	1	b_0	001	1	1	0	0	1	X	X	U
3	a_1	001	1	b_1	001	1	1	1	0	1	X	X	U
4	a_2	001	1	b_2	001	1	1	1	0	1	X	X	U
5	"	"	"	...	"	"	"	"	"	"	"	"	"
6	a_N	XXX	X	b_N	001	1	1	1	0	1	X	X	U
7	XX	XXX	X	XX	XXX	X	1	1	0	1	X	X	U
8	XX	XXX	X	XX	XXX	X	1	1	0	1	X	X	U
9	XX	XXX	X	XX	XXX	X	X	X	X	X	1	1	U
10	XX	XXX	X	XX	XXX	X	X	X	X	X	X	X	y

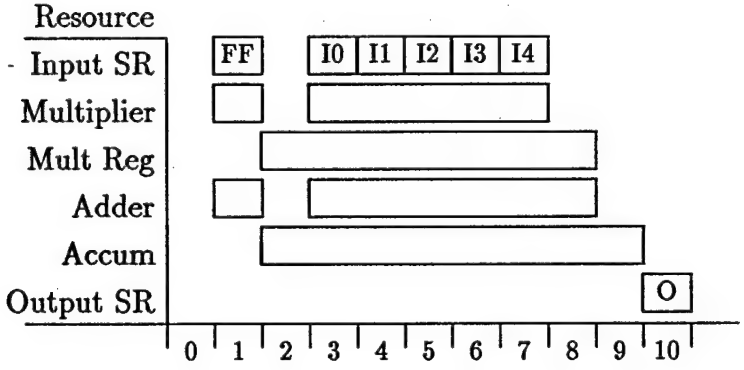


Figure 3.14: Pipeline Operation of Multiply-Accumulate Operation

3.3.3 Convolution

There are two types of discrete convolution that can be performed using the ASAP device: linear convolution and circular convolution. The linear convolution of a discrete sequence x_i of length M (x_i is zero for all $i < 0$ or $i \geq M$) and y_i of length

N (y_i is zero for all $i < 0$ or $i \geq N$) is given as

$$(x * y)(n) = \sum_{i=0}^{M+N-1} x_i y_{n-i}, \tag{3.1}$$

for all $n \in \{0, 1, 2, \ldots, M + N - 1\}$. The circular convolution of two finite discrete sequences of length N, x_i and y_i for $i \in \{0, 1, 2, \ldots, N - 1\}$, is given as

$$(x \circ y)(n) = \sum_{i=0}^{N-1} x_i y_{\langle n-i \rangle_N}. \tag{3.2}$$

First, consider the problem of mapping linear convolution to the ASAP device. Let $M = N = 3$ for purposes of illustration. Table 3.12 shows the sums of products necessary to compute $(x * y)(n)$ for $n \in \{0, 1, 2, 3, 4\}$. Each column of the table contains the product terms that must be accumulated to compute $(x * y)(n)$ for each n. In each row of the table the index of the sequence x_i is fixed: in the top row, x_0 is used for all of the product terms, in the next row, x_1 is used for all of the product terms, and in the final row x_2 is used for all of the product terms. From row to row the y_i's are seen to shift. Since $y_i = 0$ for $i \notin \{0, 1, 2\}$, several of the product terms are zero.

Table 3.12: Linear Convolution for $M = N = 3$

$n = 0$	$n = 1$	$n = 2$	$n = 3$	$n = 4$
x_0y_0	x_0y_1	x_0y_2	0	0
0	x_1y_0	x_1y_1	x_1y_2	0
0	0	x_2y_0	x_2y_1	x_2y_2
$(x * y)(0)$	$(x * y)(1)$	$(x * y)(2)$	$(x * y)(3)$	(x*y)(4)

The linear convolution computation must begin with all accumulators used initialized to zero. One set of input shift registers must be initialized with the sequence $\{y_0, y_1, y_2, 0, 0\}$. Next, x_0 is broadcast, multiply-accumulate is enabled, and the bus containing the y operands is shifted right with the shift input being zero. This pro-

cess continues for x_1 and x_2. After the appropriate pipeline delay, the results may be sampled using the Y output shift register and shifted out of the array. The procedure for linear convolution is illustrated in Table 3.13.

Table 3.13: Linear Convolution Procedure for $N = 3$

N	AB	ABS	ASREN	BB	BBS	BSREN	ENABLE	ASEL	BSEL	PSEL	YDS	YSREN	YSO
0	FF	001	1	FF	001	1	X	X	X	X	X	X	U
1	XX	XXX	0	XX	XXX	0	1	0	0	1	X	X	U
2	FF	FFF	1	y_2	001	1	1	0	0	1	X	X	U
3	FF	FFF	1	y_1	001	1	1	1	0	1	X	X	U
4	x_0	FFF	1	y_0	001	1	1	1	0	1	X	X	U
5	x_1	FFF	1	FF	001	1	1	1	0	1	X	X	U
6	x_2	FFF	1	FF	001	1	1	1	0	1	X	X	U
7	XX	XXX	X	XX	XXX	X	1	1	0	1	X	X	U
8	XX	XXX	X	XX	XXX	X	1	1	0	1	X	X	U
9	XX	XXX	X	XX	XXX	X	X	X	X	X	1	1	U
10	XX	XXX	X	XX	XXX	X	X	X	X	X	0	1	z_0
11	XX	XXX	X	XX	XXX	X	X	X	X	X	0	1	z_1
12	XX	XXX	X	XX	XXX	X	X	X	X	X	0	1	z_2
13	XX	XXX	X	XX	XXX	X	X	X	X	X	0	1	z_3
14	XX	XXX	X	XX	XXX	X	X	X	X	X	X	X	z_4

The linear convolution procedure illustrated in Table 3.13 consists of three parts: initialization, computation, and recovery of results. In steps zero and one initialization occurs. Data for the computation is shifted in steps two through six. Pipeline delays associated with the completion of the computation occur over steps seven and eight. Results are recovered in steps nine through fourteen. The pipeline operation of two $M = N = 3$ linear convolutions is illustrated in Figure 3.15. The total computational latency for linear convolution is $2(M+N)+2$ cycles from initialization to final output, however, multiple linear convolutions may be pipelined so that a sustained throughput of one linear convolution every $M + N + 1$ cycles can be achieved.

Now, consider the problem of mapping circular convolution to the ASAP device. Let $N = 3$ for purposes of illustration. Table 3.14 shows the steps necessary to compute $(x \circ y)(n)$ for $n \in \{0, 1, 2\}$. Each column of the table contains the product terms that must be accumulated to compute $(x \circ y)(n)$ for each n. In each row of the table the index of the sequence y_i is fixed: in the top row y_0 is used for all of the product terms, in the next row y_1 is used for all of the product terms, and in the last row y_2 is used for all of the product terms. Each row uses each x_i for $i \in \{0, 1, 2\}$

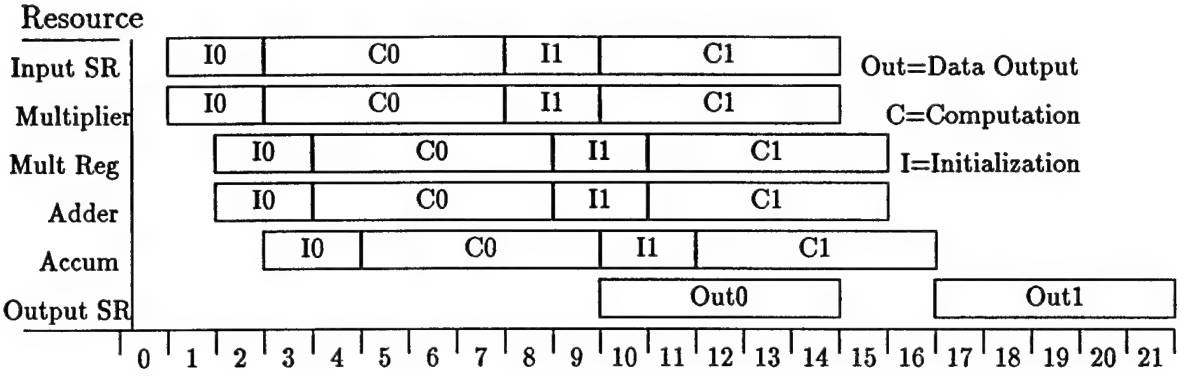


Figure 3.15: Pipeline Operation of Linear Convolution Operation for $M = N = 3$

exactly once. From row to row the x_i's are seen to circularly shift one column to the right.

The circular convolution computation must begin with all accumulators used initialized to zero. Likewise the shift registers must also be initialized to zero. The computation can begin by shifting x_2 into a shift register (the B shift register, for example) and broadcasting y_1 to all processor elements (via the A shift register). Next x_1 is shifted and y_2 is broadcast, then x_0 is shifted and y_0 is broadcast, then zero is shifted and y_1 is broadcast, and a final zero is shifted and y_2 is broadcast. The actual dataflow in this circular convolution implementation is illustrated in Table 3.15. After the appropriate pipeline delays the result of the circular convolution can be shifted out of the output shift registers. The procedure for this is illustrated in Table 3.16, and the step numbers correspond to those in Table 3.15.

Table 3.14: Circular Convolution for $N = 3$

$n = 0$	$n = 1$	$n = 2$
x_0y_0	x_1y_0	x_2y_0
x_2y_1	x_0y_1	x_1y_1
x_1y_2	x_2y_2	x_0y_2
$(x \circ y)(0)$	$(x \circ y)(1)$	$(x \circ y)(2)$

Table 3.15: Actual Dataflow for Circular Convolution for $N = 3$

Processor	$n = 0$	$n = 1$	$n = 2$
Step 2	x_0y_0	0	0
Step 3	x_2y_1	x_0y_1	0
Step 4	x_1y_2	x_2y_2	x_0y_2
Step 5	0	x_1y_0	x_2y_0
Step 6	0	0	x_1y_1
Sums	$(x \circ y)(0)$	$(x \circ y)(1)$	$(x \circ y)(2)$

Table 3.16: Circular Convolution Procedure for $N = 3$

N	AB	ABS	ASREN	BB	BBS	BSREN	ENABLE	ASEL	BSEL	PSEL	YDS	YSREN	YSO
0	FF	001	1	FF	001	1	X	X	X	X	X	X	U
1	XX	XXX	0	XX	XXX	0	1	0	0	1	X	X	U
2	y_1	FFF	1	x_2	001	1	1	0	0	1	X	X	U
3	y_2	FFF	1	x_1	001	1	1	1	0	1	X	X	U
4	y_0	FFF	1	x_0	001	1	1	1	0	1	X	X	U
5	y_1	FFF	1	FF	001	1	1	1	0	1	X	X	U
6	y_2	FFF	1	FF	001	1	1	1	0	1	X	X	U
7	XX	XXX	X	XX	XXX	X	1	1	0	1	X	X	U
8	XX	XXX	X	XX	XXX	X	1	1	0	1	X	X	U
9	XX	XXX	X	XX	XXX	X	X	X	X	X	1	1	U
10	XX	XXX	X	XX	XXX	X	X	X	X	X	0	1	z_0
11	XX	XXX	X	XX	XXX	X	X	X	X	X	0	1	z_1
12	XX	XXX	X	XX	XXX	X	X	X	X	X	X	X	z_2

The circular convolution procedure illustrated in Table 3.16 consists of three parts. The initialization part begins with the shift registers in step zero and goes into the LRNS processor in steps one and two. The computation portion begins in step two with data input to the shift registers, and is finished with the shift registers in step six, and with the LRNS processor in step eight. A snapshot of the output results is captured via the assertion of YDS and YSREN in step nine. The results are shifted out from YSO in steps ten, eleven, and twelve.

It is clear from examining Table 3.16 that the circular convolution operation is amenable to pipelining. Resource usage versus time steps for two circular convolution operations with $N = 3$ (as in Table 3.16) is shown in Figure 3.16. The total computational latency from first initialization input to final output is $3N + 4$ cycles, however, multiple circular convolutions can be pipelined so that a sustained throughput of one circular convolution every $2N + 1$ cycles can be achieved.

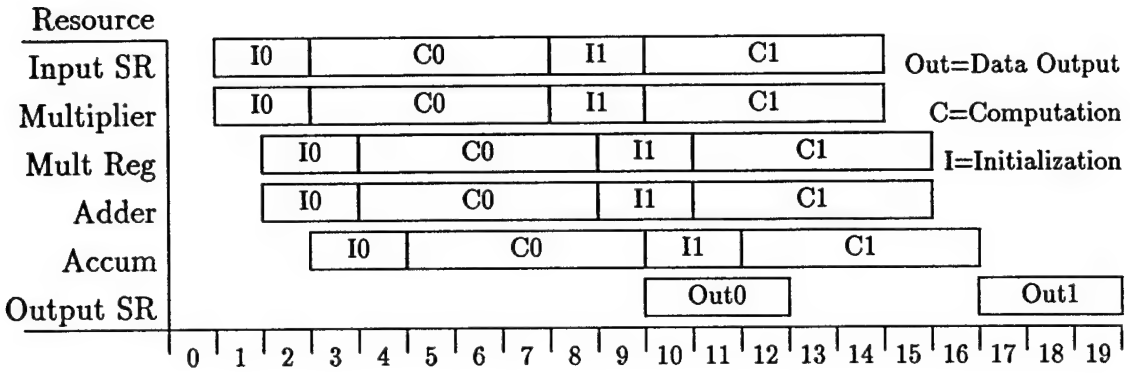


Figure 3.16: Pipeline Operation for Circular Convolution

Comparing the linear and circular convolution procedures given in Tables 3.13 and 3.16 it is seen that the two procedures are nearly identical. The primary difference is that the linear convolution procedure produces two more results than the circular convolution procedure, thus, requiring two additional cycles to shift the results. This does not impact pipelining, as can be seen by comparing pipelined operation of the linear and circular convolutions as illustrated in Figures 3.15, and 3.16.

3.4 ASAP Test Fixture

The ASAP test fixture is a solder-wrapped prototype card. The card was designed for direct connection to a Hewlett-Packard 16500A logic analysis mainframe populated with 16510B 100/35 MHz logic analyzer cards and 16520A 50 MHz pattern generator cards. The fixture provides buffering of the TTL (transistor-transistor logic) levels of the pattern generator to the 5V CMOS levels required by the ASAP chip's I/O ring. All address, data, and command signals except the read/write control signals are sampled by the logic analyzer with the comparator voltage threshold adjusted to 2.5V from the TTL preset so as to improve the analyzer's noise margin in the face of the full-swing (0V to 5V) CMOS logic levels used by the ASAP device and its data buffers. Provision is made for clocking the pattern generator, logic analyzer and ASAP chip with either a canned oscillator fed through a tapped delay line or a strobe

provided by the pattern generator, also fed through the same tapped delay line. The tapped delay line is formed with a CMOS buffer and is provided to allow the skew of the I/O to be controlled with respect to the ASAP clock. A block diagram of the card is shown in Figure 3.17. A photograph of the ASAP device in the test fixture is shown in Figure 3.18.

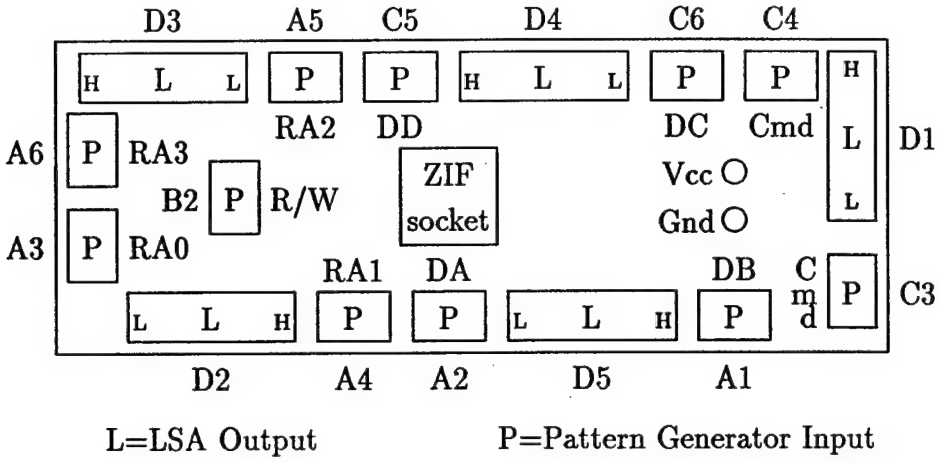


Figure 3.17: Block Diagram of ASAP Test Fixture

The pattern generator and LSA (logic state analyzer) are connected to the test board according to Table 3.17, which references Figure 3.17. The two command bytes from the pattern generator (H,L) are sampled by LSA pod D1 according to Table 3.18.

Table 3.17: Pattern Generator Pod Mapping

Pod	Signals	Pod	Signals
A6	$RA3_{7-0}$	A5	$RA2_{7-0}$
A4	$RA1_{7-0}$	A3	$RA0_{7-0}$
C5	DD_{7-0}	C6	DC_{7-0}
A2	DA_{7-0}	A1	DB_{7-0}
C4	Command(H)	C3	Command(L)
B2	R/W Control		

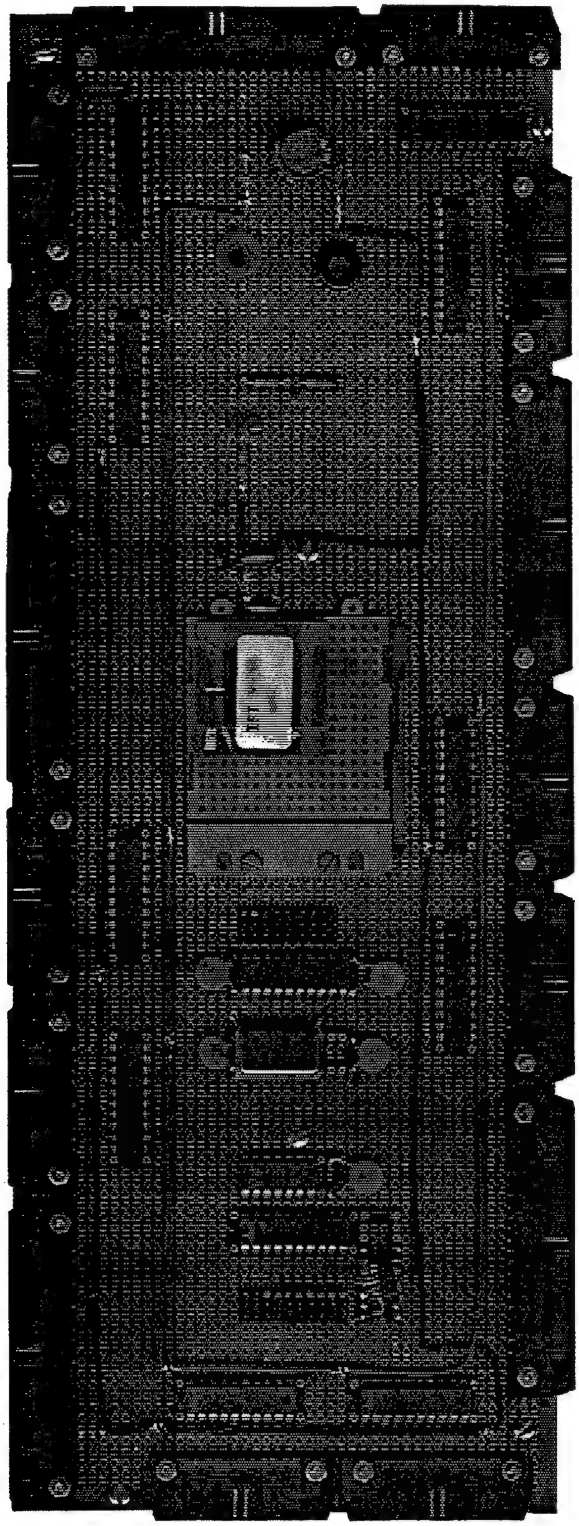


Figure 3.18: Photograph of ASAP Test Fixture with Device Under Test

Table 3.18: Command Signals to LSA D1 Pod Mapping

Pin	Signal	Pin	Signal
15	NC	7	BSEL
14	NC	6	ASEL
13	NC	5	ENABLE
12	NC	4	YSREN
11	NC	3	BSREN
10	RAMEN	2	ASREN
9	YDSEL	1	CMDREN
8	PSEL	0	DRW

3.5 ASAP Testing

The planned testing procedure consisted of three basic steps:

1. static or I_{DDQ} testing,

2. low speed functional verification, and

3. speed verification.

The initial static test was successful in eliminating those devices with fatal manufacturing defects from consideration for functional verification.

The low-speed functional verification was performed using a clock speed of 20 MHz so as to prevent any critical path timing considerations from confounding the functional verification. Functional tests were attempted using the various procedures developed previously. Using the available set of ASAP devices a basic functional test of the device was performed that verified that the processor core works. Unfortunately, full device characterization was not possible due to some errors that were uncovered during testing. With some simple modifications of the device, full characterization of the device should be possible.

3.6 Summary

While the fabricated ASAP device passed preliminary functional verification, full characterization of the device was not possible. Simulation indicates that the expected clock rate of the 0.8 μm device should exceed 100 MHz. Estimated performance metrics of the LRNS implementations in various MOSIS technologies (and beyond) are summarized in Table 3.19.

Table 3.19: Estimated Performance of LRNS MAC Cell in MOSIS Technologies, Where Available

Technology	2.0 μm (Orbit)	1.0 μm (HP)	0.8 μm (HP)	0.6 μm (HP)	0.35 μm
Area (mm^2)	1.568	0.312	0.200	0.112	0.038
Clock Freq (MHz)	40	80	100	133	230

Given the performance metrics of Table 3.19 it is reasonable to project that LRNS based signal processing solutions can span a range of arithmetic performance reaching up to 10^5 million (or more) arithmetic operations per second using currently available technology and conventional die sizes. Performance figures for arrays of thirty-two bit LRNS processors implemented in various technologies as described in Table 3.19 are summarized for both real and complex arithmetic in Table 3.20. The "equivalent real MAC rate" indicates the performance required of a conventional processor to match the quoted MAC rate.

Table 3.20: Estimated Performance of an LRNS Array of Thirty-Two Bit MACs on a 1 cm^2 Die for Real and Complex Arithmetic

Technology	2.0 μm (Orbit)	1.0 μm (HP)	0.8 μm (HP)	0.6 μm (HP)	0.35 μm
Num 32b MACs	16	80	125	223	658
Real MAC Rate (million)	640	6400	22300	87514	151340
Complex MAC Rate (million)	320	3200	11150	43757	75670
Equivalent Real MAC Rate (million)	1280	12800	44600	175028	302680

The performance estimates given in Table 3.20 are dependent upon adequate data I/O to prevent the processors from stalling. In practice, it is likely that it will only be possible to achieve such high performance figures for applications that are "highly processed." In other words, the data I/O requirements must be substantially less than the available computational bandwidth. To appreciate the impact of the I/O limitation consider the following scenario. The current upper limit for the number of pins on an integrated circuit is about 500 pins. Suppose that 256 of these pins could be used for operand inputs and that the inputs could be operated at 200 MHz. This means that eight thirty-two bit operands could enter the device per cycle, with 200 million cycles per second for an aggregate data input rate of 1.6 billion operands per second. Given that a MAC operation consumes two operands per computation cycle and assuming that one operand is stored on-chip, a 0.35 μm device as suggested in Table 3.20 would have a compute budget of nearly one hundred operations per input operand! Having said this, the number of pins that could be dedicated to input on a 1 cm^2 die (as premised in Table 3.20) is probably overstated as is the data input frequency. It likely that an actual compute budget would range into the hundreds of multiply-accumulate operations per cycle.

CHAPTER 4
VERY LONG INSTRUCTION WORD DIGITAL SIGNAL PROCESSORS

4.1 VLIW Processor Overview

The distinguishing feature of VLIW processor architecture is that each processor instruction may cause micro-operations to be issued to multiple functional units. The functional units operate in lock-step, with no additional requirements for micro-operation scheduling hardware. As a consequence, there is no run-time operation scheduling requirements. The entire burden for scheduling instructions and micro-operations occurs at the time of software compilation.

A VLIW processor for digital signal processing requires multiple functional units. These units include

- instruction fetch, decode, and issue,
- arithmetic/logic units,
- data address and fetch units, and
- operand memories.

It may also be useful to include DMA controllers as an additional functional unit to manage off-processor data transfers. This is particularly true for many digital signal processing applications where large arrays of data are processed. Autonomous DMA processors that are able to perform asynchronous transfers can greatly simplify access to external memories that have variable access times and transfer rates. A

block diagram illustrating general structure of a candidate VLIW DSP processor is shown in Figure 4.1.

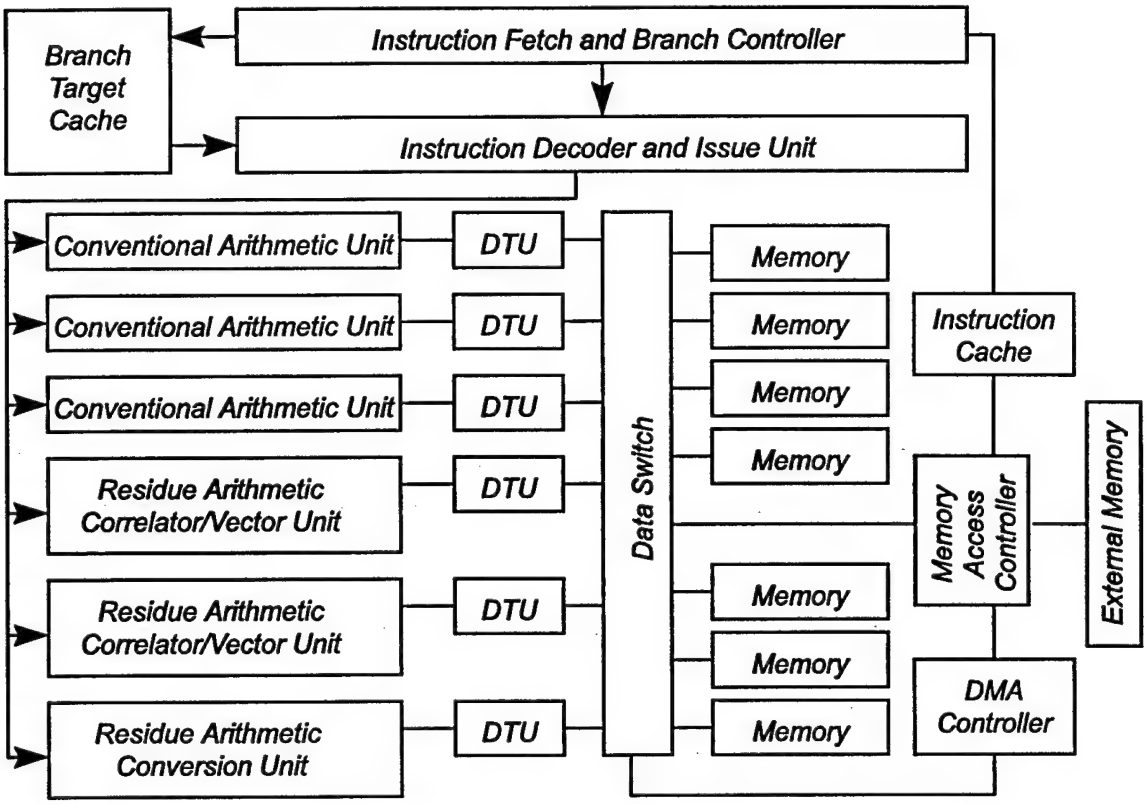


Figure 4.1: VLIW Machine Architecture Block Diagram

The architecture shown in Figure 4.1 has several distinguishing features. First, all of the arithmetic functional units are coupled to local (on-chip) memory blocks via a switch. Arithmetic operations are performed only using operands obtained from these local memories. The motivation for this restriction is to isolate computations from the long, possibly variable (certainly unknowable at compilation time) delays associated with external (off-chip) memory accesses. If external memory access time is unknown or variable then allowing programmed access to external memory could result in processor stalls or possibly gross code expansion. The impact of either of these consequences would be application dependent. The architecture shown is essentially a load-store architecture, except rather than using register files for storage

as in general purpose processors, much larger memories are used. A substantial benefit of the load-store architecture presented here is that the many address arithmetic units, embedded in the DTUs (data-transfer units) shown in Figure 4.1, are substantially smaller than they would be if they were required to be capable of addressing the relatively vast external memory space.

Since the majority of processing in DSP applications is performed upon arrays of data, a DMA controller is provided to transfer data between internal and external memories. An independent DMA controller may be employed to perform block memory transfers, isolating the processor from the impact of variable memory access times. Synchronization of block transfers performed by an independent DMA controller is substantially less expensive than the word-by-word synchronization that would be required by programmed data transfers.

The local memory blocks depicted in Figure 4.1 would take form of small SRAMs with one or more read/write ports. The optimal size of the on-chip memories is application dependent. Access to the memories is mediated by a data switch. For a small processor with few functional units a single-level monolithic switch is ideal, however, for a large-scale processor a hierarchical switch architecture may offer better performance and lower cost by partitioning data traffic between the arithmetic functional units and their associated local memories between non-dependent parallel micro-operation streams. This possibility is explored in a more quantitative manner in Section 5.3.1.

The instruction fetch and branch decoder depicted in Figure 4.1 must be capable of dealing with variable length instructions due to instruction compaction that must occur in order to manage instruction bandwidth. Instruction fetch from external memories should be mediated by an instruction cache; the value of an instruction

cache is, in many cases, even greater for DSP applications than for those applications typically executed on general purpose computers.

The functional units required for a VLIW DSP processor are explored in greater detail in the next section.

4.2 VLIW Processor Functional Units

This section describes the features associated with each of the major functional units illustrated in Figure 4.1.

4.2.1 Instruction fetch and decode unit

The instruction fetch and decode unit in a VLIW architecture is potentially somewhat more complicated than that found in traditional RISC and CISC processors. The source of this complication stems from the immense instruction bandwidth required by a VLIW processor: on each instruction cycle there may be a micro-operation issued for each functional unit. In contrast, in a traditional RISC or CISC architecture only a small number of micro-operations may be issued each instruction cycle. To provide the requisite number of micro-operations for a VLIW architecture, an extremely long instruction word (e.g., 256 bits could, conceivably, be required for a four-way VLIW architecture) may be required. As suggested in Section 3.6, the input bandwidth of any implementation is limited, so it is important to address the issue of instruction bandwidth.

An extremely long instruction word produces at least two significant challenges. First, since it is unlikely that each functional unit will be issued a non-NOP (no-operation) micro-instruction on each instruction cycle, a reasonable means of compacting the instruction must be determined. In other words, many VLIW instructions may be inherently low-entropy, and therefore, the required raw instruction bandwidth will be much greater than that required by a processor with a high-entropy instruc-

tion stream (e.g., a CISC instruction stream). The second problem is raised by the first. Assume that some form of instruction compaction is introduced to increase the entropy of the stored instructions. Then the instructions are inherently variable length. If the instructions have variable length and are significantly compacted then instruction decoding is complicated. A complicated instruction format may cause instruction decoding to become a performance bottleneck. To address this problem a balance must be struck between instruction compaction efficiency and fetch and decoding efficiency.

The easiest way to achieve the balance between compaction efficiency and decoding simplicity is to include with each instruction one bit per encoded micro-instruction indicating whether that micro-instruction is an NOP [24]. If the micro-instruction is flagged as an NOP then it is not included in the instruction word. The instruction decoder must then expand the instruction based upon the NOP flags. This method of compaction is illustrated in Figure 4.2. The fetch unit must be capable of fetching compacted instructions the cross memory word boundaries. The fetch unit may determine the number of machine words that must be fetched to assemble one compacted instruction by decoding the NOP flags. It is important that the individual micro-instruction have fixed length. Additional NOP flags may, however, be used to indicate micro-instruction extensions, such as immediate operands.

4.2.2 Address arithmetic unit

Address arithmetic for DSP is more complicated than that found in general purpose microprocessors. In addition to linear array indexing, modular (circular) and bit-reversed array addressing modes are desirable in DSP. Furthermore, many existing DSP microprocessors use dedicated address registers with dedicated address

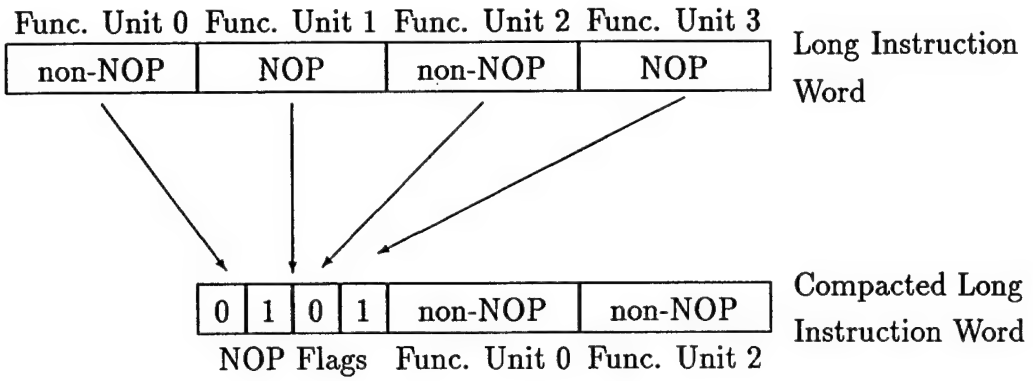


Figure 4.2: Example of VLIW Instruction Compaction

arithmetic units capable of supporting these operations. A block diagram of an address arithmetic unit suitable for DSP operations is shown in Figure 4.3.

The structure shown in Figure 4.3 can support the set of addressing modes summarized in Table 4.1. This structure is the primary component of the DTU shown in Figure 4.1. The structure includes a register file, which is used either directly or indirectly to store the address arithmetic parameters (index, modulus, stride, and base

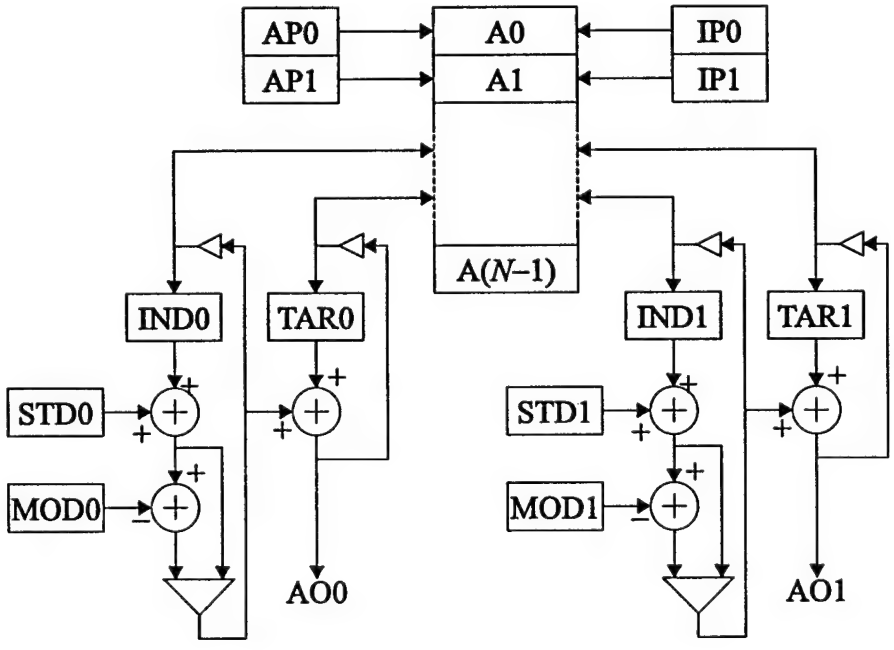


Figure 4.3: Block Diagram of an Address Arithmetic Unit

address). At least two arithmetic units would be required to support sum-of-products operations. Vector operations (e.g., point-wise addition) would require at least three address arithmetic units unless the result is overwriting a vector operand.

Table 4.1: Addressing Modes Supported by Address Arithmetic Unit

Addressing Mode	Address Computation
AR Indirect	(TAR)
AR Indirect Indexed	(TAR+IND)
AR Indirect Indexed, Linear Index Post-Incremented	(TAR+IND) IND $\leftarrow$ IND + STD
AR Indirect Indexed, Circular Index Post-Incremented	(AR+IND) IND $\leftarrow$ IND + STD mod MOD
AR Indirect Indexed, Bit-reversed Index Post-Incremented	(AR+IND) IND $\leftarrow$ IND $\vec{+}$ STD

4.2.3 Conventional arithmetic unit

Conventional arithmetic functional units for a VLIW DSP processor take the same form found in traditional DSP microprocessors — namely, multiplier-accumulator units. Both fixed-point and floating-point units are appropriate for use in VLIW DSP processors. Subdivisions of large datapaths into smaller (word length) datapaths that are operated in a SIMD manner on packed data (e.g., two sixteen bit words packed into a thirty-two bit word) are also appropriate for DSP applications.

4.2.4 Residue arithmetic units

Residue arithmetic multiplier-accumulator. A block diagram of an enhanced version of the multiplier-accumulator in the ASAP device is shown in Figure 4.4. This extended arithmetic unit offers two significant features that were not present in the arithmetic unit used in the ASAP device, namely a logic unit and a second accumulator.

A standalone RNS multiplier-accumulator unit probably offers little advantage over a conventional arithmetic multiplier-accumulator as a functional unit for a VLIW

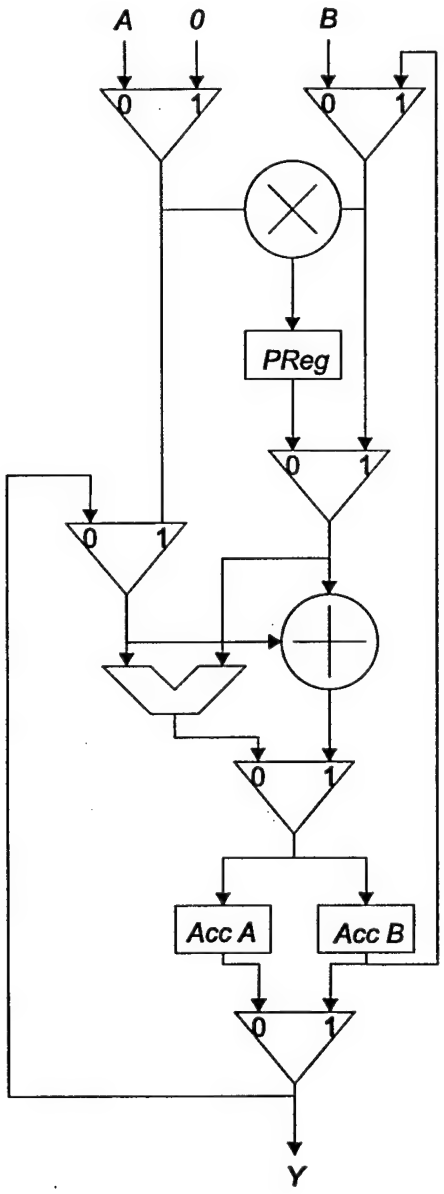


Figure 4.4: Extended RNS MAC Architecture

digital signal processor where conventional arithmetic units are present. The advantages of RNS processors can be best exploited in a functional unit that uses multiple devices, such as a convolver or correlator.

Residue arithmetic vector unit. A significant problem that was identified in the ASAP implementation was the long length of the correlator structure. Since the

lengths of the convolutions that had to be performed to support the desired transform length varied widely, overall processor utilization was not optimal. To increase overall processor utilization for shorter convolution lengths, a shorter correlator structure is proposed in Figure 4.5.

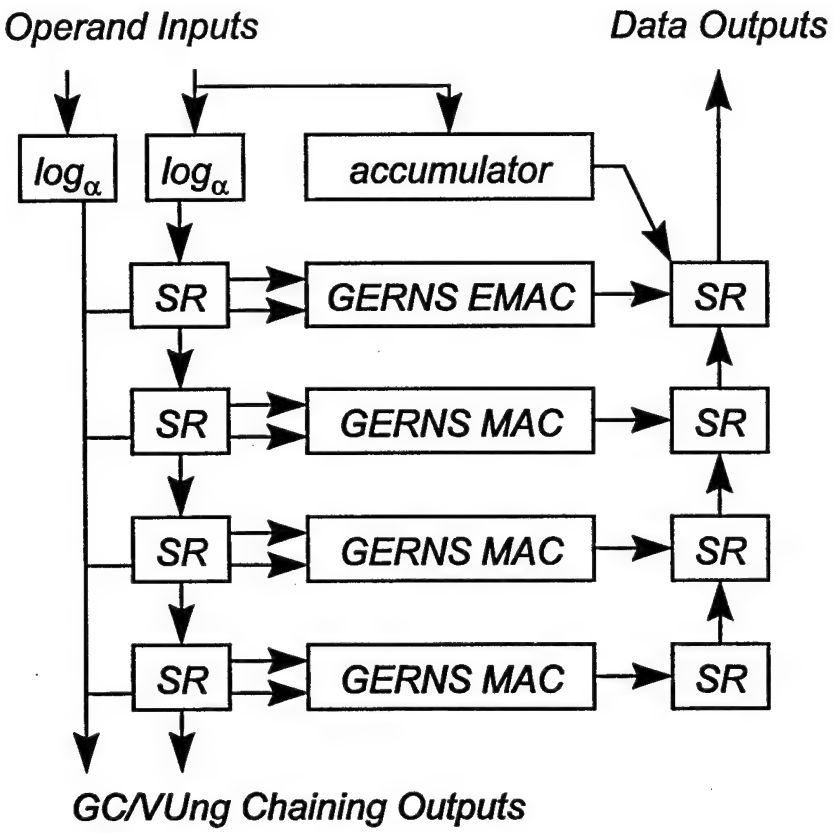


Figure 4.5: Next Generation Vector Unit

By itself, the four multiplier-accumulator vector unit shown in Figure 4.5 can easily be used to perform a Rader prime DFT of up to length five. To support greater correlation lengths, chaining may be used to append adjacent vector units to form a larger vector unit. For example, in the case of a Good-Thomas FFT of length $3 \times 7 \times 11 = 231$, the constituent Rader prime DFTs may be performed using one (unchained) vector unit for those transforms of length three, two chained vector

units for those transforms of length seven, and three chained vector units for those transforms of length eleven.

The advantages of this correlator structure are fairly obvious. Supplying one or two operands per operation cycle, the unchained unit shown in Figure 4.5 can achieve up to four multiply-accumulate operations. Therefore, the vector unit provides a means of achieving relatively high arithmetic bandwidth versus the number of operands supplied per operation cycle. With chaining, even higher operation bandwidths may be achieved without increasing the operand bandwidth.

Residue arithmetic data conversion unit. Residue arithmetic conversion is a necessary function in a DSP processor environment that includes residue arithmetic elements. There are two possible approaches to meeting this need. The first is to place forward conversion elements on the inputs to the arithmetic elements, and backward conversion elements on the outputs of the arithmetic elements. This may be inefficient because residue arithmetic data may be recirculated, resulting in unnecessary backward/forward conversion steps. Another reason why transparent data conversion may be undesirable is because it may result in unnecessary *repetitive* conversion of fixed coefficient data.

An alternative to transparent conversion of RNS data is to convert the data by explicitly using a separate or loosely integrated conversion function unit. The advantage of this approach is that conversion is only performed when required. This may substantially reduce the amount of conversion performed, and may reduce the minimum required number of conversion units compared to the case where transparent conversion is performed. The disadvantage of this approach is precisely its advantage, the conversion must be explicitly managed in the the instruction stream.

4.3 On-Chip Memories

On-chip, processor-local memories are a critical component of a VLIW digital signal processor. The reasons for this are manifold; compared to on-chip memories, off-chip chip memories have

- much lower bandwidth,
- long access latencies, and
- possibly variable access latencies.

A further incentive to minimize off-chip memory accesses is the greater energy required to access an off-chip memory. Not only does wasted power impact battery life in mobile applications, but it may also substantially increase packaging expenses.

In general purpose processors, local memories take the form of register files and cache memories. As previously stated, since DSP applications operate on arrays of data, it is more useful to supply processor-local data memory instead of register files or cache memories. This is, in fact, consistent with classic vector supercomputers with vector registers [2].

On-chip memories may be arranged in two formats. One possible means of arranging on-chip memories is in one global memory block with multiple banks or ports and access mediated either through a non-blocking or blocking switch or bus resource. In this model, generally referred to as a uniform memory access (UMA) model, all of the memory is uniformly accessible by all functional units. The UMA model provides maximal flexibility and the simplest possible resource scheduling.

An alternative to the UMA model is a non-uniform memory access (NUMA) model. In the NUMA model some memories are preferentially associated with specific processor resources. The advantage of the NUMA model is that it allows for greater scalability (i.e., more functional units) with greater theoretical peak performance

and lower cost compared to the UMA model. The disadvantage is that scheduling processor resources in an NUMA environment is more difficult than in an UMA environment due to the memory access constraints implied by the NUMA model.

CHAPTER 5
VERY LONG INSTRUCTION WORD COMPILER TECHNOLOGY

5.1 Introduction

Since VLIW processors have no hardware instruction scheduling capabilities, it is incumbent upon the compiler to perform instruction scheduling. The advantages of performing instruction scheduling at compilation time are substantial. First, there is no recurring (i.e., per processor) cost for instruction scheduling. Instructions scheduled at compilation time should be more efficiently scheduled than possible at run-time since the compiler has more complete information about the program than the processor has — both in the sense of having access to the program source code, and having a complete view of the object code for final instruction scheduling.

Since VLIW processors have multiple functional units they can be expected to be able to exploit opportunities for instruction-level and block-level parallelism. Opportunities for block-level parallelism can be exploited on any processor architecture. Exploitation of block-level parallelism has always occurred at compilation time, not run-time. On the other hand, opportunities for instruction-level parallelism may be identified both at compile-time and run-time. In fact, many general purpose microprocessors dynamically reschedule instructions at run-time to best exploit opportunities for instruction-level parallelism.

To support the paradigm of a custom configured VLIW processor it is necessary to insulate the software engineer from detailed knowledge of the hardware. To do this, a new programming language has been defined: C_{DSP} . The C_{DSP} language

is significant in that like its namesake, C, C_{DSP} is a high-level assembly language for DSP applications that are executed on DSP microprocessors.

5.2 The C_{DSP} Programming Language

The C_{DSP} programming language is a high-level assembly language for DSP applications that are executed on DSP microprocessors. Its suitability transcends VLIW DSP processors; its semantic features closely match the architectural features found in many common DSP microprocessors. A detailed description of the language, in the form of a language reference manual, is contained in Appendix A. The language reference manual contains a complete description of the language, including the constituent productions of a LALR grammar for the language.

5.2.1 Motivation

To support a program first, select hardware last system integration paradigm it is necessary to allow processor independent software development. The means of achieving processor independence is to select a high-level language that can be targeted to any likely processor implementation. The high-level language of choice for high-performance application development is C. The C language provides excellent performance when used to develop applications for many general purpose computers. However, this isn't true when C is used to develop DSP applications for DSP microprocessors. The reason that the C language produces such good executable object for general purpose processors and such poor executable object for DSP microprocessors is that the language has syntactic and semantic elements that reflect the architecture of general purpose microprocessors, not DSP microprocessors. For this reason, the C language is considered to be a "high-level assembly language" for general purpose processors.

What is needed is a "high-level assembly language" for DSP microprocessors. The C language can be modified, adding language elements that reflect the needs of DSP applications and the architecture of DSP microprocessors, and removing those language elements that interfere with the emission of efficient executable object for DSP microprocessors. To this end, the C_{DSP} language has been created.

5.2.2 Differences between C and C_{DSP}

This section describes the significant differences between the C and C_{DSP} languages. There are some features that are defined in the C_{DSP} language that are not found in the C language. In particular, the C_{DSP} language has support for array operations and defines new operators for common DSP operations. The C_{DSP} language also lacks some of the features of the C language such as pointers and dynamic memory allocation.

Parallel looping. Since the C_{DSP} language was defined to allow efficient DSP application code generation for DSP microprocessors with multiple functional units, supporting both block level and instruction level parallelism, the standard parallel looping construct, `dopar`, is implemented in the C_{DSP} language. The `dopar` statement implements an efficient fork-join mechanism that is particularly useful for applications such as parallel computation of matrix multiplications. The `dopar` statement is discussed in detail in Section A.9.5.

Elimination of unneeded features. The C_{DSP} language does not have the `struct` feature found in the C language. For some DSP microprocessors, full support of the `struct` may cause some difficulty because the DSP microprocessor's addressing capabilities are highly optimized for operation upon *simple* arrays of data, not arrays of nested `structs`. If arrays of `structs` are required they may be efficiently

implemented using multiple arrays where each array corresponds to one element of the structure.

The `switch` statement found in the C language is not found in the C_{DSP} language. For the most part, the behavior of the `switch` statement can be emulated with the `if-else` statement. The `switch` statement is not found in C_{DSP} primarily to simplify compiler construction. Since most DSP applications are loop intensive, and not selection intensive, the `switch` is unlikely to be greatly missed.

The `double` intrinsic type is not found in the C_{DSP} language. The justification is that most floating-point DSP microprocessors do not have support for more than one floating-point format. Therefore, the `float` type is the only intrinsic type defined for floating-point representations.

Unlike the C language, the C_{DSP} language does not allow recursive function calls. This is done primarily for performance reasons and to simplify the compile-time dynamic memory allocation management problem. Furthermore, type types of computations required for DSP applications are generally more efficiently executed using loops rather than recursive functions. A more detailed discussion of these issues is found in Section A.4.1.

Elimination of pointers. The C_{DSP} language eliminates the pointers found in the C language. Pointers are a useful machination for many applications executed on general purpose computers, however, they interfere with dependence analysis necessary to re-order and parallelize code. This is primarily due to the fact that it is difficult to determine the value of a pointer at compile-time.

By eliminating pointers, dynamic memory allocation as found in the C language is eliminated. While dynamic memory allocation is an important element of many applications executed on general purpose computers, it is not needed for basic DSP

applications. Since DSP applications are generally single tasks that are executed on embedded processors memory is usually statically allocated.

Elimination of pointers in the C_{DSP} language also impacts the mechanism of passing arrays of data to functions. In the C language arrays are passed to functions by reference, i.e., using pointers. The C_{DSP} language maintains the passing of arrays by reference, however, since unrestricted pointers are not allowed, the actual parameter for any particular function call may be determined at compile time, even if it is passed through multiple functions.

Semantics of the increment and decrement operators. DSP applications operate primarily upon arrays of data. Arrays of data are accessed using indexes. In DSP applications arrays are frequently accessed using non-unit stride indexes. Many DSP microprocessors include hardware support for non-unit stride array indexing, modular array indexing, and bit-reversed array indexing. To provide direct support for these hardware features, the C_{DSP} language adds a new intrinsic type, `index`, and changes the semantics of the increment and decrement operators when acting on variables of type `index`. In particular, when using the `index` type the stride does not have to be unity, automatic modular indexing, and bit-reversed indexing may be performed without complicated conditional processing required on most general purpose computers, and in the C language. The details of the semantics of the increment and decrement operators in the C_{DSP} language are detailed in Section A.6.2.

Array expression operators. The C_{DSP} language modifies the semantics of most of the arithmetic operators to allow operation on array operands. Most of the binary operators have been modified so that they may operate on array operands with compatible geometry, as well as scalars and arrays. The mixing of scalar and array operands is accomplished by acting as if the scalar operand were actually a

constant array with the same geometry as the array operand and the same value in each element as the value of the scalar. The operators with array operand support are described in detail in Sections A.6.3, A.6.4, A.6.6, A.6.7, A.6.8, A.6.11, A.6.12, A.6.13, and A.6.17.

The C_{DSP} language also has several new operators to support linear and circular convolution and sums-of-products. These operations are the cornerstones of digital signal processing. Consequently, the presence of these operators has great value to the programmer, as well as to the compiler writer. For the programmer this means that these operations can be expressed very compactly. For the compiler writer, the convolution and sum-of-products operators enable the emission of compact, efficient object code. The details of the operation of these operators are detailed in Section A.6.5.

Sub-array expressions. Since it is sometimes necessary to perform arithmetic operations on sub-arrays, a means of addressing sub-arrays without explicitly copying out the sub-array is required. To support this requirement, C_{DSP} has an index range notation similar to that commonly found in other languages. This range notation is described in detail in Section A.6.2. Sub-arrays determined using index range notation are equivalent to full arrays with geometry determined by the size of the index set in each dimension.

5.2.3 Results

The C_{DSP} language has been carefully tuned to allow succinct expression of DSP algorithms, and to allow efficient emission of object for DSP microprocessors in general, and VLIW DSP microprocessors in particular.

The C_{DSP} language effects compact algorithm expression primarily by the addition of array operators to the language. Other differences between the C_{DSP} and C languages that effect not only the compactness of expression of DSP algorithms, but

also the performance of the compiled code on a DSP microprocessor are index range notation for the specification of sub-arrays, the `index` type, which is intended for indexing arrays, and the modified semantics of the increment and decrement operators when operating upon the `index` type.

The C_{DSP} language aids in the automatic generation of parallel code by eliminating language features that hinder automatic parallelization, such as unrestricted pointers, and adding parallel looping constructs such as the `dopar` statement. While an obvious casualty of the elimination of unrestricted pointers is dynamic memory allocation, the need for dynamic memory allocation on single task, embedded DSP microprocessors is somewhat less than on multi-tasking general purpose computers. Furthermore, the availability of the automatic storage class (`auto`) in the C_{DSP} language mitigates the lack of dynamic memory allocation.

The definition of the C_{DSP} language is significant in that it is a high-level language designed for embedded DSP applications executed on DSP microprocessors. It is also significant in that it is designed to enable the compiler to exploit every reasonable opportunity for block level and instruction level parallelism. The C_{DSP} language is successful as a "high-level assembly language," enabling development of efficient DSP applications for embedded DSP microprocessors. This, in turn, allows the application to be coded before the target architecture is selected. This opens the option of tailoring the processor to just fit the application. The implications of the ability to use a processor with no more hardware than absolutely required to meet the needs of the application are profound.

5.3 Algorithm Analysis

This section analyzes the cornerstone algorithm of DSP, convolution (and its applications to filtering), as well as the discrete Fourier transform, and the QR decompo-

sition. These algorithms are analyzed to quantify their amenability to parallelization by exploitation of available opportunities for block level and instruction level parallelism. This information is significant in that it determines how much benefit can be expected from VLIW digital signal processors.

5.3.1 Convolution and the finite impulse response filter

The finite linear convolution sum, used for FIR filtering, has the form

$$y_n = \sum_{k=0}^{N-1} a_k x_{n-k}, \tag{5.1}$$

where the finite sequence $\{a_0, a_1, a_2, \ldots, a_{N-1}\}$ is generally a fixed set of coefficients, $\{x_n\}$ is an input data sequence, and $\{y_n\}$ is the output data sequence. This finite sum of products on the right hand side of Equation 5.1, whether an actual convolution sum or not is the cornerstone operation in digital signal processing. As a consequence, it must be highly optimized in any processor implementation intended for DSP applications.

In a VLIW processor implementation the sum in Equation 5.1 may be partitioned among L processor elements, with a final accumulation of L partial sum-of-products taken as a final step in forming the convolution sum. Suppose that $L = 2$ and $L \mid N$ (L divides N). Then Equation 5.1 may be partitioned into the sum

$$y_n = y_{n,0} + y_{n,1}, \tag{5.2}$$

where

$$y_{n,0} = \sum_{k=0}^{N/2-1} a_k x_{n-k}, \tag{5.3}$$

```
fixed(short,15) fA[41], fX[41];
fixed(short,10) fY;
index iN,iM;
int iCount;

/* Assume that fA is initialized somewhere. */
iN.ind=iN.base=0; iN.mod=41; iN.stride=1;
iM.ind=iM.base=0; iM.mod=41; iM.stride=1;

while (1) {
  fA[iN++]=read(); /* Get new datum. */

  /* Compute filter output (convolution sum). */
  for(fY=0, iCount=0; iCount<41; ++iCount)
    fY+=fA[iM++]*fX[iN++]

  write(fY); /* Write filter output. */
}
```

Figure 5.1: C_{DSP} Source for Convolution Sum

and

$$y_{n,1} = \sum_{k=N/2}^{N-1} a_k x_{n-k}. \tag{5.4}$$

This is the obvious partitioning strategy and leads to an implementation that is illustrated in Figure 5.2.

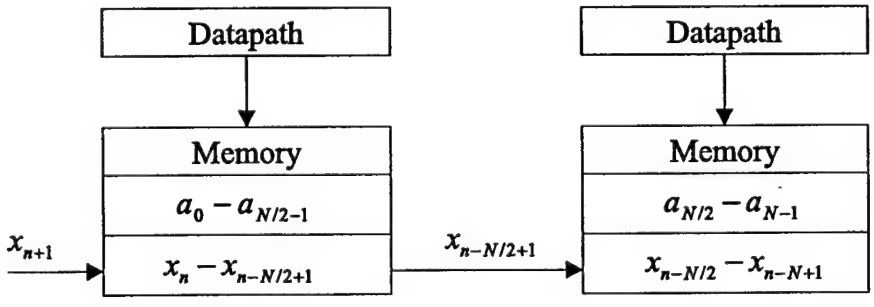


Figure 5.2: Data Distribution and Flow for Two Processor Convolution Sum

The partitioning strategy illustrated in Figure 5.2 shows that before each filter cycle, the newest datum, x_{n+1}, must be written into a local data memory and a datum must be transferred from one local memory to another. Final accumulation of the partial sums-of-products is not illustrated here.

There exists another approach to partitioning the sum of products in Equation 5.1. Again, suppose that $L = 2$ and $L \mid N$. Then Equation 5.1 may be decomposed into the sum

$$y_n = y'_{n,0} + y'_{n,1}, \tag{5.5}$$

where

$$y'_{n,0} = \sum_{k=0}^{N/2-1} a_{2k+\langle n \rangle_2} x_{n-2k}, \tag{5.6}$$

and

$$y'_{n,1} = \sum_{k=0}^{N/2-1} a_{2k+1+\langle n \rangle_2} x_{n-2k-1}. \tag{5.7}$$

This leads to the implementation illustrated in Figure 5.3.

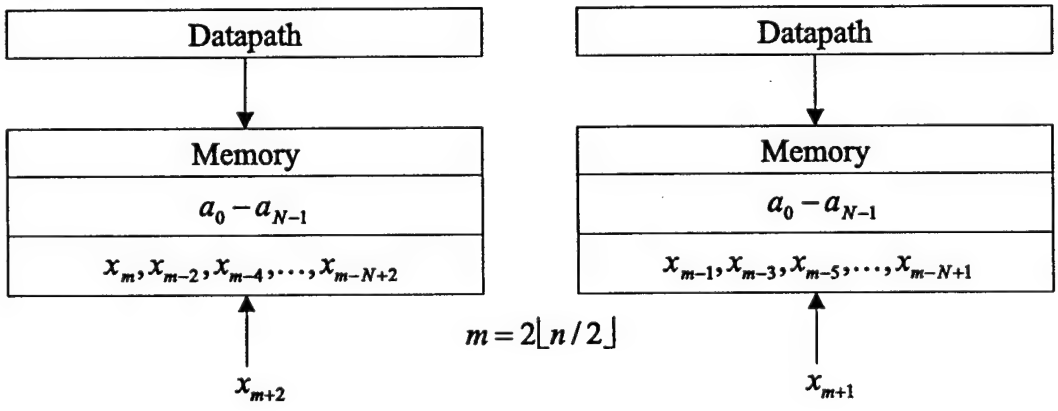


Figure 5.3: Data Distribution and Flow for Two Processor Convolution Sum Using Interleaved Data

In the implementation illustrated in Figure 5.3, each local memory contains a complete set of the coefficients $\{a_0, \ldots, a_{N-1}\}$, but only half of the data sequence $\{x_n\}$ (either the even indexed or the odd indexed elements). The differences between the implementations shown in Figures 5.2 and 5.3 are similar to a decimation-in-time versus a decimation-in-frequency fast Fourier transform implementation. The advantage of this second implementation approach is that there are no inter-local memory data transfers required so the overall global memory traffic per filter cycle is reduced. The disadvantage of this implementation strategy is the need to store all of the coefficients in each local memory.

The implementation strategies described above can be generalized to L processors. In general, suppose that $L \mid N$. Without loss of generality, if $L \nmid N$ then the sequence $\{a_0, \ldots, a_{N-1}\}$ can be padded with $\langle N \rangle_L$ zeros so that $L \mid N$. Then the sum in Equation 5.1 can be decomposed into the sum of partial sums of products

$$y_n = \sum_{p=0}^{L-1} y_{n,p} \tag{5.8}$$

where

$$y_{n,p} = \sum_{k=p\lceil N/L \rceil}^{(p+1)\lceil N/L \rceil - 1} a_k x_{n-k}. \tag{5.9}$$

The data distribution for this multiprocessor convolution sum is illustrated in Figure 5.4.

The cost parameters associated with performing a convolution sum in the manner suggested in Figure 5.4 are

$$N_{\text{MAC}} = N(\text{internal MAC cycles}), \tag{5.10}$$

$$N_{\text{acc}} = L - 1(\text{final partial sum accumulation}), \tag{5.11}$$

$$N_{\text{xfer}} = L - 1(\text{global data transfers}), \tag{5.12}$$

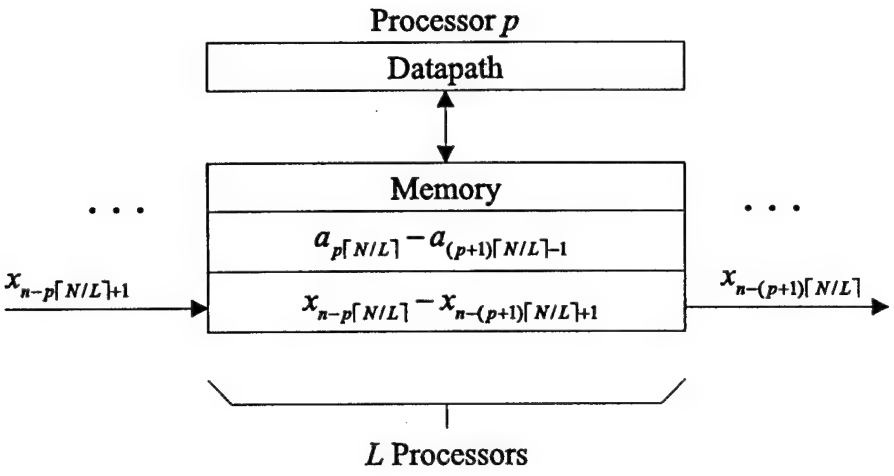


Figure 5.4: Data Distribution for an L Processor Convolution Sum

$$N_{\text{coef}} = \lceil N/L \rceil (\text{coefficient storage per processor}), \text{ and} \tag{5.13}$$

$$N_{\text{data}} = \lceil N/L \rceil (\text{data storage per processor}). \tag{5.14}$$

The execution time is given by the weighted sum

$$N_{\text{cyc}} = \alpha_{\text{MAC}} \lceil N/L \rceil + \alpha_{\text{acc}}(L-1) + \alpha_{\text{xfer}}(L-1). \tag{5.15}$$

The $L-1$ factor of the $\alpha_{\text{acc}}(L-1)$ term represents a worst case scenario for the accumulation of the partial sums of products. Depending upon the global data transfer resources it may be possible to reduce this term to $\alpha_{\text{acc}} \log_2 L$. The total memory consumption used by this approach is minimal,

$$N_{\text{memory}} = N_{\text{coef}} + N_{\text{data}} = 2N. \tag{5.16}$$

The second generalized approach, based upon the two processor case illustrated in Figure 5.3 decomposes the convolution sum shown in Equation 5.1 into the sum of

partial sums

$$y_n = \sum_{p=0}^{L-1} y'_{n,p}, \tag{5.17}$$

where

$$y'_{n,p} = \sum_{k=0}^{\lceil N/L \rceil - 1} a_{kL+p+\langle n \rangle_p} x_{n-kL-p}. \tag{5.18}$$

The data distribution suggested by Equation 5.18 is illustrated in Figure 5.5.

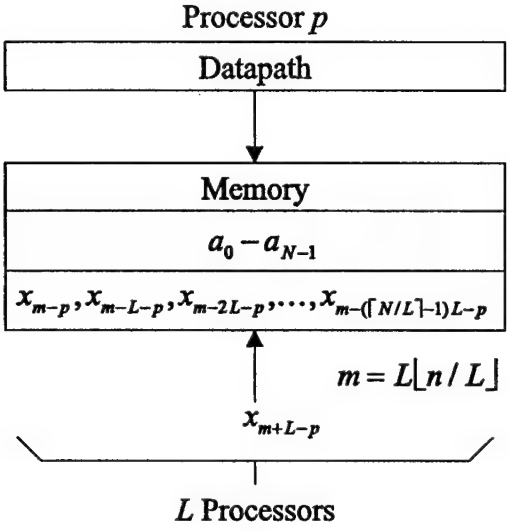


Figure 5.5: Data Distribution for an L Processor Convolution Sum Using Interleaved Data

The performance metrics for this approach are the same as those listed previously, except for

$$N_{\text{xfer}} = 1, \tag{5.19}$$

and

$$N_{\text{coef}} = NL. \tag{5.20}$$

The total execution time for this approach is given the by the weighted sum

$$N'_{\text{cyc}} = \alpha_{\text{MAC}} \lceil N/L \rceil + \alpha_{\text{acc}}(L-1) + \alpha_{\text{xfer}}, \tag{5.21}$$

where, as before, it may be possible to improve $\alpha_{\text{acc}}(L-1)$ towards the limit $\alpha_{\text{acc}} \log_2 L$. The total memory consumed by this approach is

$$N_{\text{memory}} = LN_{\text{coef}} + N_{\text{data}} = (L+1)N. \tag{5.22}$$

If N is large then the interleaved data approach may be overly memory intensive, however, the additional memory usage is mitigated by the reduction in global memory traffic compared to the direct, non-interleaved approach. The relative cost of the block decomposition versus the interleaved decomposition is dependent upon the fine architectural details which are lumped into the weights shown in Equations 5.15 and 5.21. Among the issues that impact the value of these weights and execution time are

- the number of ports and banks in each processor-local memory block,
- interconnection resources,
- L, and
- N.

The whole point of distributing a sum of products computation among multiple processors is to obtain a speedup. Without identifying a specific architecture a best case speedup (versus a single processor) is given by

$$\text{Speedup} = \frac{N}{\lceil N/L \rceil + \log_2(\min(N, L))}, \tag{5.23}$$

where N is the filter order and L is the number of processors used. The results of this equation for $N \in \{5, 10, 15, 20, 25, 30, 35, 40\}$ and $L \in \{1, 2, 3, \ldots, 20\}$ are shown in Figure 5.6. From the graph it can be seen that the application of additional

processors can produce a speedup — up to a point. After the maximum speedup is achieved, additional processors can actually reduce the speedup. The reduction in speedup caused by the additional processors is a result of increased time spent in accumulating the final sum of the partial sums of products. It is also clear from the plot that the maximum speedup is highly dependent upon the filter order, increasing as the filter order increases.

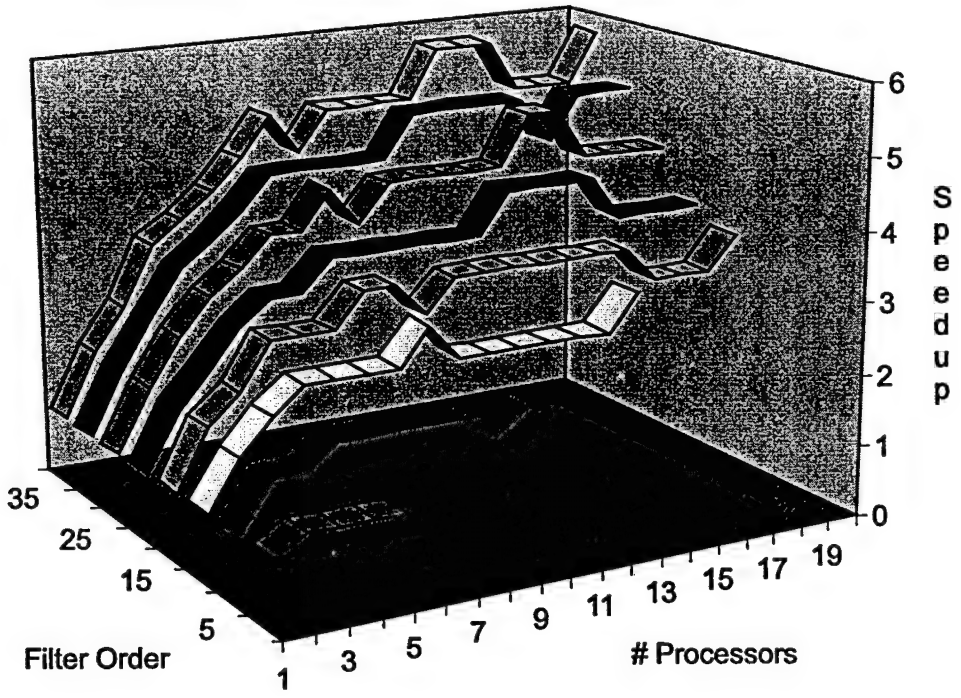


Figure 5.6: VLIW Filter Speedup Versus Filter Order and Number of Processors, Best Case

To highlight the negative impact that too many processors may have, consider modifying Equation 5.23 to make interprocessor communication more expensive,

$$\text{Speedup} = \frac{N}{\lceil N/L \rceil + \min(N, L) - 1}. \tag{5.24}$$

The results of this over the same values of N and L as used to create Figure 5.6 are shown in Figure 5.7. The impact of applying too many processors is even more

pronounced in this case. It is also worth noting that the maximum speedup that can be achieved for any particular filter order is significantly lower than that suggested by Equation 5.23.

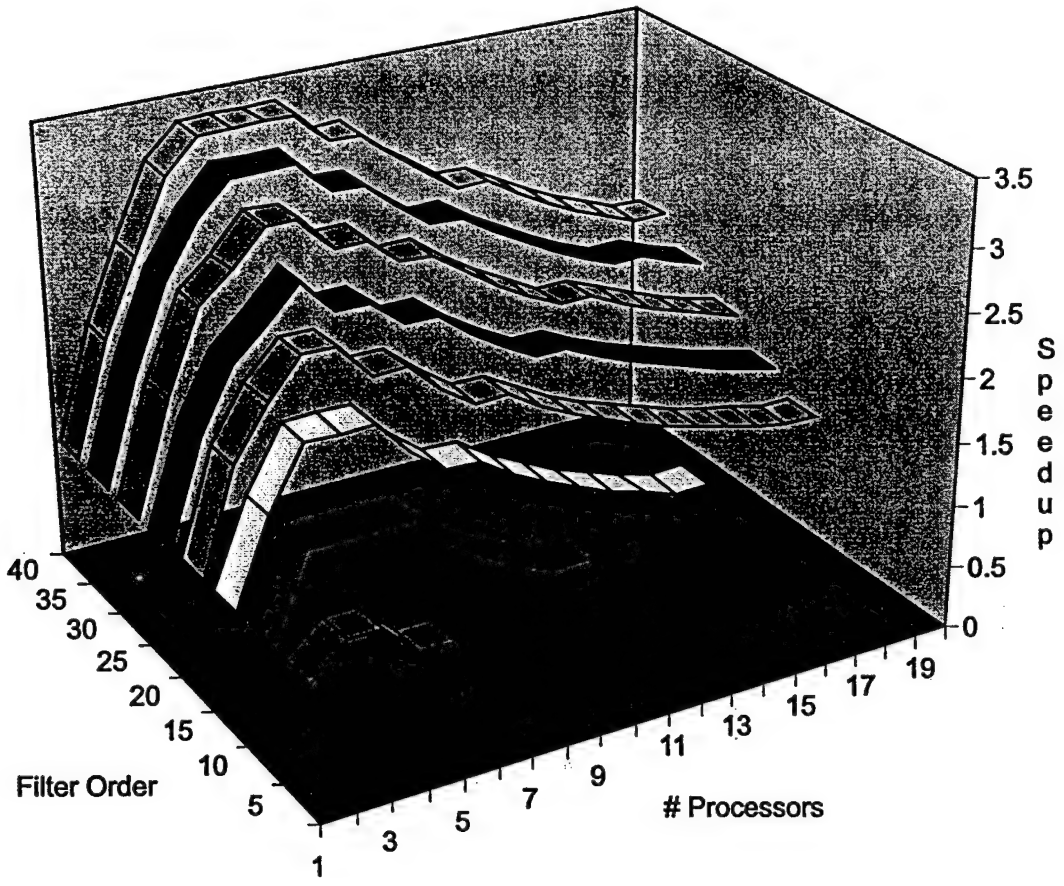


Figure 5.7: VLIW Filter Speedup Versus Filter Order and Number of Processors, Worst Case

If one assumes that the likely values of N are bounded then it is clear from the data shown in Figures 5.6 and 5.7 that there is an upper bound, much less than N, to the number of processors that can usefully applied to a particular sum of products. This suggests that given a large number of processor elements, a hierarchical NUMA architecture with three or more levels of access would provide worthwhile benefits. For instance, the data in Figure 5.6 suggests that not more than eight processors can be efficiently applied to a sum of products. Therefore, it would make sense to

take a block of eight processors with local memories, add a processor-memory switch that is confined to that group and a global switch. This is illustrated in Figure 5.8. The L1 interconnect is a direct connection between a single processor and a single memory. The L2 interconnect is a switch that allows direct access between processors and memory within the group (i.e., intra-group connectivity). The L3 interconnect is a global switch that allows connection of processors and memories outside not in the same group (i.e., intergroup connectivity).

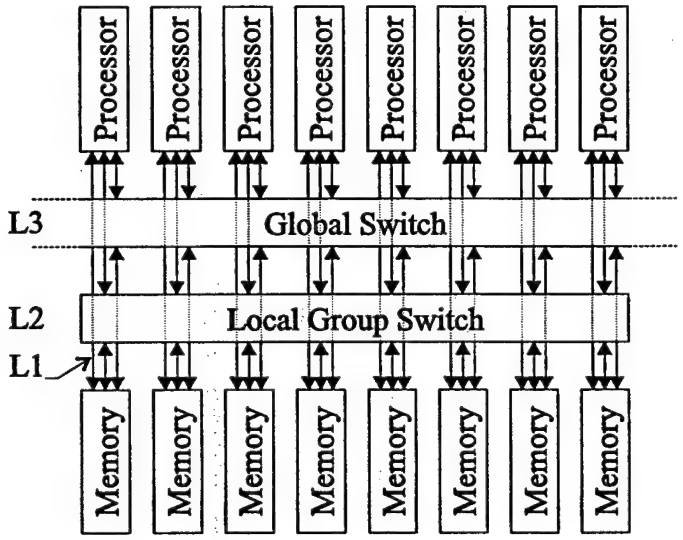


Figure 5.8: Group of Processor Elements with Three-Level Hierarchical Processor/Memory Switching

The optimal granularity of grouping in a three or more level hierarchical interconnect scheme would be highly dependent upon the number of functional units in the processor and the characteristics of the anticipated applications. To evaluate the effects of a three level hierarchical NUMA scheme, Equation 5.24 may be modified to reflect parallel local interprocessor communications and serial intergroup communications by

$$\text{Speedup} = \frac{N}{\lceil N/L \rceil + \min(N, L, G) + \lfloor \min(N, L)/G \rfloor}, \tag{5.25}$$

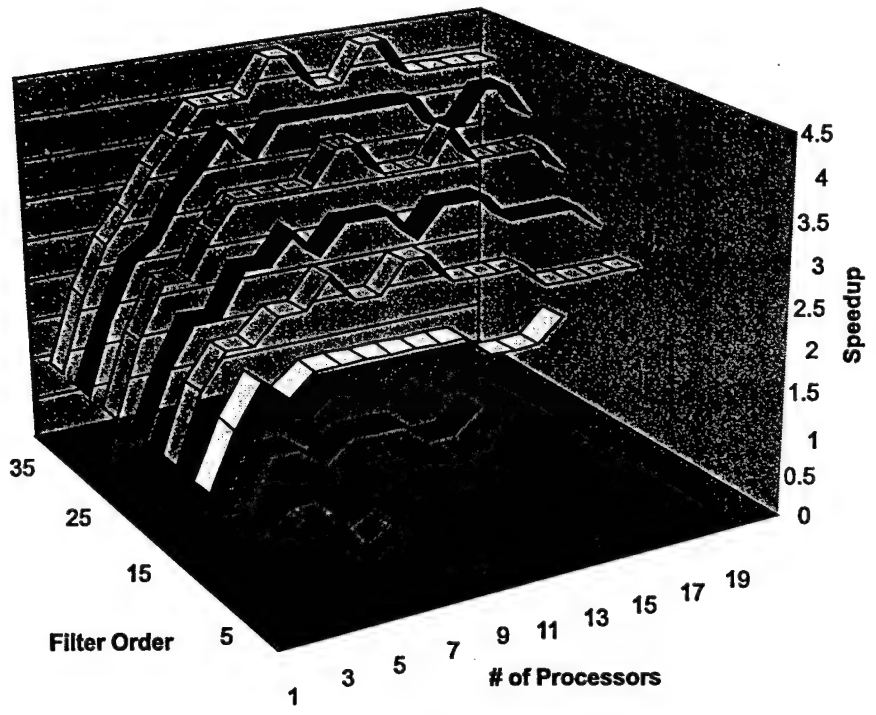
where G is the processor grouping factor (i.e., the number of processors bound by an L2 interconnect, see Figure 5.8 where $G = 8$). In particular, the first denominator term reflects the parallel computation of a sum of products, the second term reflects intragroup (L2) communication, and the third term reflects intergroup (L3) communication. Evaluating Equation 5.25 over the same values of N and L used to create Figures 5.6 and 5.7 with grouping factors $G = 4$ and $G = 8$ produces the results shown in Figure 5.9.

The speedup curves, are similar to those produced assuming global non-blocking communications shown in Figure 5.6, although the peak speedup is not as great. However, the speedups are greater than that shown in Figure 5.7. In Figure 5.9, the results for $G = 4$ are seen to result in greater peak spedup than those shown for $G = 8$. This is balanced by the fact that global interconnect is used with twice the frequency when $G = 4$ compared to when $G = 8$. Clearly, a balance must be struct between intragroup and intergroup communications.

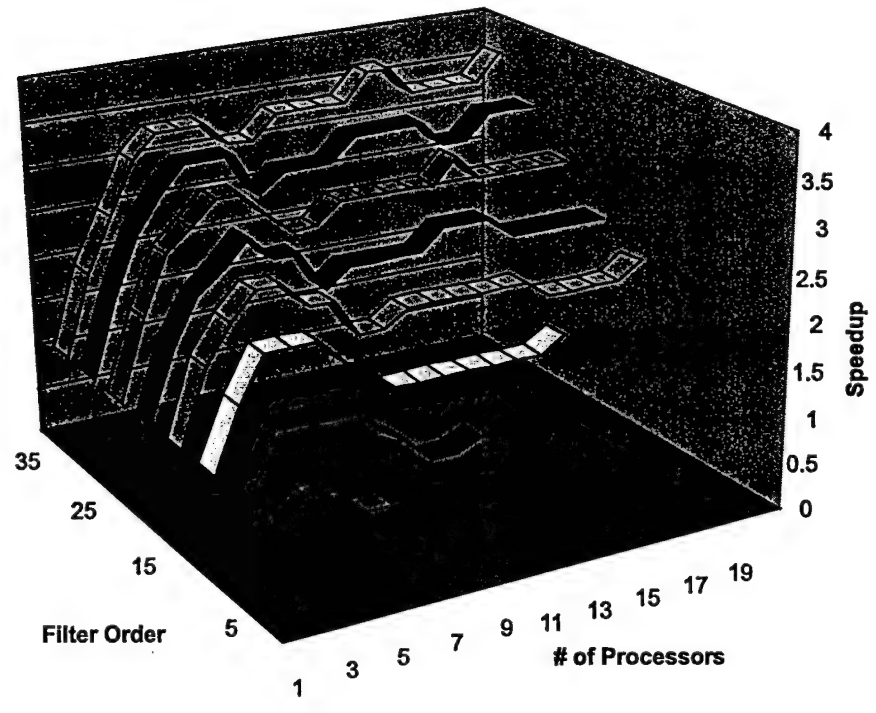
5.3.2 Discrete Fourier transform

The discrete Fourier transform is one of the most significant DSP functions. Real-time, high-speed implementations of the DFT are increasingly important, driven by new applications in video processing (compression) and communications (digital subscriber loop technologies). The Good-Thomas and Rader prime DFTs are described here since these algorithms lead to efficient hardware implementations, specifically, the ASAP device was designed to execute these algorithms, performing 256×256-class DFTs at video rates.

Good-Thomas DFT. The Good-Thomas DFT [7, 9] is an efficient algorithm for computing the DFT of a sequence of length M where M is composite. Let $M = \prod_{i=1}^{L} p_i$ where $\gcd(p_i, p_j) = 1$ for all $i, j \in \{1, 2, 3, \ldots, L\}$ and $i \neq j$. Define $m_i = M/p_i$



(a)



(b)

Figure 5.9: VLIW Filter Speedup Versus Filter Order and Number of Processors Using NUMA Interconnect with (a) $G = 4$, and (b) $G = 8$

and let m_i^{-1} denote the multiplicative inverse of m_i in $\mathbb{Z}_{p_i}$, that is, $m_i m_i^{-1} \equiv 1 \pmod{p_i}$. The Chinese remainder theorem describes an isomorphism

$$\phi : \mathbb{Z}_M \longrightarrow \mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \mathbb{Z}_{p_3} \times \cdots \times \mathbb{Z}_{p_L} \tag{5.26}$$

where $\phi(X) = (x_1, x_2, x_3, \ldots, x_L)$, and each $x_i \equiv X \pmod{p_i}$ for all $i \in \{1, 2, 3, \ldots, L\}$. The inverse mapping is given as

$$\phi^{-1}((x_1, x_2, x_3, \ldots, x_L)) = \left\langle \sum_{i=1}^{L} m_i \langle m_i^{-1} x_i \rangle_{p_i} \right\rangle_M . \tag{5.27}$$

The DFT of an M point sequence $\{x_n\}$ is given as $X_k = \sum_{n=0}^{M-1} x_n \omega^{nk}$ where $\omega = e^{-j2\pi/M}$. By the CRT, let $\phi(n) = (n_1, n_2, n_3, \ldots, n_L)$. Define a mapping

$$\psi : \mathbb{Z}_M \longrightarrow \mathbb{Z}_{p_1} \times \mathbb{Z}_{p_2} \times \mathbb{Z}_{p_3} \times \cdots \times \mathbb{Z}_{p_L} \tag{5.28}$$

with

$$k = \psi^{-1}(k_1, k_2, k_3, \ldots, k_L) = \langle m_1 k_1 + m_2 k_2 + m_3 k_3 + \cdots + m_L k_L \rangle_M . \tag{5.29}$$

Substituting into the DFT produces

$$\begin{aligned} X_{\psi^{-1}((k_1,k_2,k_3,\ldots,k_L))} &= \sum_{n=0}^{M-1} x_n \omega^{n\psi^{-1}((k_1,k_2,k_3,\ldots,k_L))} \\ &= \sum_{n_1=0}^{p_1-1} \cdots \sum_{n_L=0}^{p_L-1} x_{\phi^{-1}((n_1,n_2,n_3,\ldots,n_L))} \omega^{\phi^{-1}((n_1,\ldots,n_L))\psi^{-1}((k_1,\ldots,k_L))} . \end{aligned} \tag{5.30}$$

Since ω is of order M (it is the Mth primitive root of unity in $\mathbb{C}$),

$$\omega^{\phi^{-1}((n_1,\ldots,n_L))\psi^{-1}((k_1,\ldots,k_L))} = \omega^{\sum_{i=1}^{L} (m_i \langle m_i^{-1} n_i \rangle_{p_i})(m_i k_i)} . \tag{5.31}$$

As each $m_i = M/p_i$ and $\omega = e^{-j2\pi/M}$,

$$\begin{aligned}
\omega^{(m_i\langle m_i^{-1}n_i\rangle_{p_i})(m_ik_i)} &= e^{-j2\pi m_i^2\langle m_i^{-1}n_i\rangle_{p_i}k_i/M} && (5.32)\\
&= e^{-j2\pi m_i\langle m_i^{-1}n_i\rangle_{p_i}k_i/p_i} && (5.33)\\
&= e^{-j2\pi m_i m_i^{-1}n_ik_i/p_i}\\
&= e^{-j2\pi n_ik_i/p_i}\\
&= \omega^{m_in_ik_i}.
\end{aligned}$$

This leads to

$$\begin{aligned}
X_{\psi^{-1}((k_1,k_2,k_3,\ldots,k_L))} &= \sum_{n_1=0}^{p_1-1}\cdots\sum_{n_L=0}^{p_L-1} x_{\phi^{-1}((n_1,n_2,n_3,\ldots,n_L))}\omega^{m_1n_1k_1+\cdots+m_Ln_Lk_L} && (5.34)\\
&= \sum_{n_1=0}^{p_1-1}\omega^{m_1n_1k_1}\left[\cdots\left[\sum_{n_L=0}^{p_L-1} x_{\phi^{-1}((n_1,\ldots,n_L))}\omega^{m_Ln_Lk_L}\right]\cdots\right].
\end{aligned}$$

This is clearly an L-dimension DFT and may be computed in $M\sum_{i=1}^{L}p_i$ complex multiply-accumulates.

The result shown in Equation 5.35 appears to be quite complicated. In fact, its application is relatively simple. To illustrate this, consider an $M = 3\times 5 = 15$ Good-Thomas FFT. The permutations described by Equations 5.27 and 5.29 for $p_1 = 3$ and $p_2 = 5$ produce the permutation maps shown in Figure 5.10.

The significance of the permutation maps shown in Figure 5.10 are that they show the way to an efficient implementation. If the sequence to be transformed is $\{x_0, x_1, x_2, \ldots, x_14\}$, then the first step is to map this sequence into a two dimensional array according to the map shown for ϕ^{-1}. While the map for ϕ^{-1} appears complicated, in fact, by tracing the locations of the sequence $\{0, 1, 2, \ldots, 14\}$ a simple pattern is apparent. The result of this mapping is shown in Figure 5.11. Once the

$\phi^{-1}(n_1, n_2)$

n_2 \ n_1	0	1	2
0	0	10	5
1	6	1	11
2	12	7	2
3	3	13	8
4	9	4	14

$\psi^{-1}(k_1, k_2)$

k_2 \ k_1	0	1	2
0	0	5	10
1	3	8	13
2	6	11	1
3	9	14	4
4	12	2	7

Figure 5.10: Good-Thomas FFT Permutation Maps for $M = 3 \times 5 = 15$

input sequence is mapped to the two-dimensional array, length three DFTs may be performed on each row followed by length five DFTs on each column (or vice versa).

$\phi^{-1}(n_1, n_2)$

n_2 \ n_1	0	1	2
0	x_0	x_{10}	x_5
1	x_6	x_1	x_{11}
2	x_{12}	x_7	x_2
3	x_3	x_{13}	x_8
4	x_9	x_4	x_{14}

$\psi^{-1}(k_1, k_2)$

k_2 \ k_1	0	1	2
0	X_0	X_5	X_{10}
1	X_3	X_8	X_{13}
2	X_6	X_{11}	X_1
3	X_9	X_{14}	X_4
4	X_{12}	X_2	X_7

Figure 5.11: Good-Thomas FFT Input/Output Sequence Permutation for $M = 15$ Computation

After performing the row-wise and column-wise DFTs, the final results may be recovered according to the permutation map for ψ^{-1}, with the locations of the results in the two-dimensional array illustrated in Figure 5.11. As before, the permutations required to recover the results appear complicated, but are relatively simple. By following the locations of the elements of the sequence $\{X_0, X_1, X_2, \ldots, X_{14}\}$ in order, a simple pattern is apparent.

A C_{DSP} function to implement the Good-Thomas FFT for a fifteen element real array is given in Figure 5.12. The function takes two arrays as parameters — one

containing the real input data. Both arrays are used to return the real and imaginary parts of the result. The function begins by permuting the original real data into a three column by five row array. Next, five length three DFTs are performed on the rows of the array followed by three length five DFTs that are performed on the columns of the array. The DFTs are done within two dopar loops, taking advantage of the C_{DSP} language's mechanism for allowing the programmer to identify opportunities for parallelism. The form of the DFTs shown is that of a direct DFT computed by matrix multiplication with a twiddle matrix. A more efficient means of computing the prime length DFTs required for the Good-Thomas FFT is the Rader prime DFT. The C_{DSP} function shown in Figure 5.14 demonstrates the C_{DSP} code that would be inserted into the Good-Thomas FFT function of Figure 5.12. The final step in the Good-Thomas FFT function is to extract the results.

The Good-Thomas FFT is attractive for VLSI implementation due to the efficient way in which the required small prime block length DFTs can be computed using the Rader prime DFT discussed in the following section. Using just the primes in $\{2, 3, 5, 7, 11, 13\}$ Good-Thomas FFTs of fifty different composite lengths between six and 30030 can be computed. These lengths are summarized in Table 5.1.

Rader prime DFT. While the radix-two FFT is well known for efficient operation, the butterfly structure introduces unnecessary complexity in a VLSI implementation. An alternative algorithm known as the Rader prime algorithm [10, 7] is available. The Rader prime algorithm performs the DFT using cyclic convolution which is very amenable to a full custom VLSI implementation.

Let the block length of the DFT be p, a prime. Then there exists some α such that α generates $GF(p) \setminus \{0\}$ (i.e., α is a primitive element of $GF(p)$). Define a

```
const fixed(long,10)fT5R[5][5]={
```

Insert twiddle matrix defined by $\mathrm{Re}(W_{m,n}) = \mathrm{Re}(e^{-j2\pi mn/5})$.

```
  };
const fixed(long,10)fT5I[5][5]={
```

Insert twiddle matrix defined by $\mathrm{Im}(W_{m,n}) = \mathrm{Im}(e^{-j2\pi mn/5})$.

```
  };
const fixed(long,10)fT3R[3][3]={
```

Insert twiddle matrix defined by $\mathrm{Re}(W_{m,n}) = \mathrm{Re}(e^{-j2\pi mn/3})$.

```
  };
const fixed(long,10)fT3I[3][3]={
```

Insert twiddle matrix defined by $\mathrm{Im}(W_{m,n}) = \mathrm{Im}(e^{-j2\pi mn/3})$.

```
  };

void GTFFT(fixed(long,10)fXRe[15], fixed(long,10)fXIm[15])
{ fixed(long,10) fXMRe[5][3], fXMIm[5][3], fDR[5], fDI[5];
  index iM,iN; int iL;

  /* Permute original real data. */
  iM.mod=5; iM.stride=1; iM.base=0;
  iN.mod=3; iN.stride=1; iN.base=0;
  for (iM.ind=iN.ind=iL=0; iL<15; iM++, iN++, iL++) {
    fXMRe[iM][iN]=fXRe[iL];
    fXMIm[iM][iN]=0.0; /* Original data is assumed real. */
  }

  /* Perform length 3 DFTs on rows. */
  dopar (iM.ind=0; iM<5; ++iM) {
    for (iN.ind=0; iN<3; ++iN) {
      fDR[iN]=fXMRe[iM][0:2]$$fT3R[iN][0:2];
      fDI[iN]=fXMRe[iM][0:2]$$fT3I[iN][0:2];
    }
    fXMRe[iM][0:2]=fDR[0:2]; fXMIm[iM][0:2]=fDI[0:2];
  }
```

Figure 5.12: C_{DSP} Function for an $N = 15$ Good-Thomas FFT

```
  /* Perform length 5 DFTs on columns. */
  dopar (iN.ind=0; iN<3; ++iN) {
    for (iM.ind=0; iM<5; ++iM) {
      fDR[iN]=fXMRe[0:4][iN]$$fT5R[0:4][iN]-
              fXMIm[0:4][iN]$$fT5I[0:4][iN];
      fDI[iN]=fXMRe[0:4][iN]$$fT5I[0:4][iN]+
              fXMIm[0:4][iN]$$fT5R[0:4][iN];
    }
    fXMRe[0:4][iN]=fDR; fXMIm[0:4][iN]=fDI;
  }

  /* Extract results. */
  iM.stide=2; iN.stride=2;
  for (iM.ind=iN.ind=iL=0; iL<15; ++iM, ++iN, ++iL) {
    fXRe[iL]=fXMRe[iM][iN]; fXIm[iL]=fXMIm[iM][iN];
  }
}
```

Figure 5.12 – continued

permutation

$$\phi(n) = \alpha^n, \tag{5.35}$$

for all $n \in \{1, 2, 3, \ldots, p-1\}$. The DFT of a sequence f_n is given as

$$\begin{aligned} F_n &= \sum_{k=0}^{p-1} f_k e^{-j2\pi nk/p} \\ &= f_0 + \sum_{k=1}^{p-1} f_k e^{-j2\pi nk/p}. \end{aligned} \tag{5.36}$$

Substituting in the permutation rule of Equation 5.35 produces

$$F_{\phi(\phi^{-1}(n))} = f_0 + \sum_{k=1}^{p-1} f_{\phi(\phi^{-1}(k))} e^{-j2\pi\phi(\phi^{-1}(n))\phi(\phi^{-1}(k))/p} \tag{5.37}$$

Table 5.1: Product of All Combinations of Two or More Primes in {2,3,5,7,11,13}

Primes	Product	Primes	Product
2,3	6	3,5,13	195
2,5	10	2,3,5,7	210
2,7	14	3,7,11	231
3,5	15	3,7,13	273
3,7	21	2,11,13	286
2,11	22	2,3,5,11	330
2,13	26	5,7,11	385
2,3,5	30	2,3,5,13	390
3,11	33	3,11,13	429
5,7	35	5,7,13	455
3,13	39	2,3,7,11	462
2,3,7	42	2,3,7,13	546
5,11	55	5,11,13	715
5,13	65	2,3,11,13	858
2,3,11	66	7,11,13	1001
2,5,7	70	3,5,7,11	1155
7,11	77	3,5,7,13	1365
2,3,13	78	2,3,5,7,11	2310
7,13	91	2,3,5,7,13	2730
3,5,7	105	2,3,5,11,13	4290
2,5,11	110	5,7,11,13	5005
2,5,13	130	2,3,7,11,13	6006
2,7,11	154	2,5,7,11,13	10010
3,5,11	165	3,5,7,11,13	15015
2,7,13	182	2,3,5,7,11,13	30030

$$= f_0 + \sum_{k=1}^{p-1} f_{\phi(\phi^{-1}(k))} e^{-j2\pi\phi(\phi^{-1}(n)+\phi^{-1}(k))/p},$$

for $n \in \{1, 2, 3, \ldots, p-1\}$, with $F_0 = \sum_{k=0}^{p-1} f_k$. Let $q = \phi^{-1}(n)$ and $r = p - \phi^{-1}(k)$. Then

$$\begin{aligned} F_{\phi(q)} &= f_0 + \sum_{k=1}^{p-1} f_{\phi(p-(p-\phi^{-1}(k)))} e^{-j2\pi\phi(q-p+\phi^{-1}(k))/p} \\ &= f_0 + \sum_{k=1}^{p-1} f_{\phi(p-r)} e^{-j2\pi\phi(q-r)/p}. \end{aligned} \tag{5.38}$$

Now, set $F'_q = F_{\phi(q)}$ and $f'_r = f_{\phi(p-r)}$. Then

$$F'_q = f_0 + \sum_{r=0}^{p-2} f'_r e^{-j2\pi\phi(p-r)/p}, \tag{5.39}$$

which is clearly the form for circular convolution. A block diagram of an architecture to perform the Rader prime DFT is shown in Figure 5.13. A Matlab function, *rpdft*, is provided in Section B.1.1, which computes the DFT of a prime length sequence using the Rader prime algorithm.

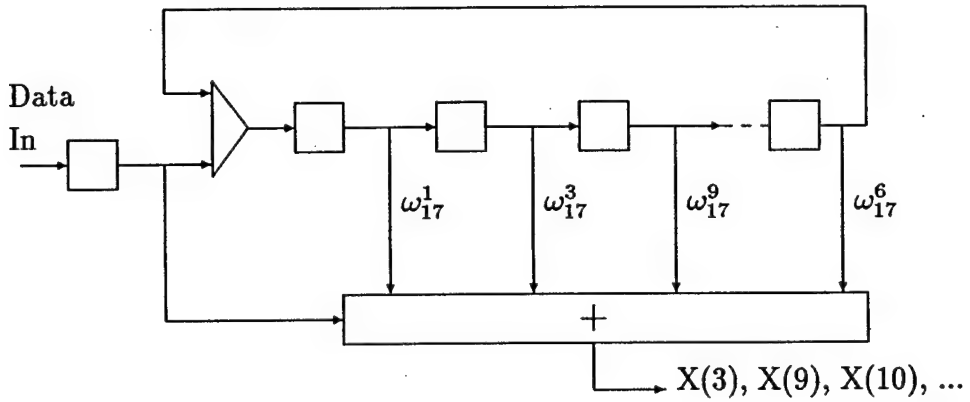


Figure 5.13: Rader Prime DFT Circular Convolution Engine, $p = 17$

A C_{DSP} implementation of a $p = 5$ Rader prime DFT is shown in Figure 5.14. The function starts by permuting four elements of the two parameters by direct assignment. Next, the circular convolution required for the DFT is performed using predefined permuted twiddle factor arrays. Finally, the X_0 component is compiled and the results of the circular convolution are permuted and placed into the parameter arrays in natural order. There are some limited opportunities for parallelism in this function, primarily in the computation of the circular convolution operations and the X_0 term. A VLSI implementation of a DFT may see benefits from the RNS. In particular, Zelniker and Taylor [12], have demonstrated that an RNS based VLSI implementation of the Rader prime DFT can be easily achieved.

```
const fixed(long,10) fTR[4]={
      Insert {Re(ω5^1),Re(ω5^2),Re(ω5^4),Re(ω5^3)}.
  };
const fixed(long,10) fTI[4]={
      Insert {Im(ω5^1),Im(ω5^2),Im(ω5^4),Im(ω5^3)}.
  };

void RPDFT5(fixed(long,10)fXR[5], fixed(long,10)fXI[5])
{ fixed(long,10) fYR[4], fYI[4], fZR[4], fZI[4];

  /* Permute input data. */
  fYR[0]=fXR[1];  fYI[0]=fXI[1];
  fYR[1]=fXR[2];  fYI[1]=fXI[2];
  fYR[2]=fXR[4];  fYI[2]=fXI[4];
  fYR[3]=fXR[3];  fYI[3]=fXI[3];

  /* Rader prime DFT circular convolution. */
  fZR=fXR[0]+(fYR @ fTR - fYI @ fTI);
  fZI=fXI[0]+(fYR @ fTI + fYI @ fTR);

  /* Compute X_0 and permute results. */
  fXR[0]=1$$fXR;  fXI[0]=1$$fXI;

  fXR[1]=fZR[0];  fXI[1]=fZI[0];
  fXR[2]=fZR[1];  fXI[2]=fZI[1];
  fXR[4]=fZR[2];  fXI[4]=fZI[2];
  fXR[3]=fZR[3];  fXI[3]=fZI[3];
}
```

Figure 5.14: C_{DSP} Implementation of a $p = 5$ Rader Prime DFT

5.3.3 QR decomposition

The QR decomposition [5] is an important tool in digital signal processing, particularly in spectrum estimation, adaptive filtering, and beamforming applications [6, 36]. The QR factorization theorem is stated as follows.

Theorem 5.1 (QR factorization) *If* $\mathbf{A} \in \mathbb{C}^{m \times n}$ *is of rank* n*, then* $\mathbf{A}$ *can be factored into a product* $\mathbf{QR}$ *where* $\mathbf{Q} \in \mathbb{C}^{m \times n}$ *is a matrix with orthonormal columns, and* $\mathbf{R} \in \mathbb{C}^{n \times n}$ *is upper triangular and invertible.*

The QR decomposition enables the robust solution of linear algebraic equations of the form

$$\mathbf{Ax} = \mathbf{b}. \tag{5.40}$$

Approaches such as Gaussian elimination are not as robust as the QR decomposition.

The author has previously developed the implementation requirements for a QR decomposition in a vector processing residue arithmetic environment [37]. There are essentially two basic implementation strategies for the QR decomposition: one relies upon Householder reflections while the other relies upon Givens rotations. The Householder reflection approach is preferred by those using vector machines while the Givens rotation approach is preferred by those using parallel machines. A VLIW DSP has attributes of both vector processors and parallel processors, however, the Givens rotation approach requires a substantial number of square root and division operations [5, p. 202]. In contrast, the Householder reflection is multiply-accumulate intensive with only one square root and one scalar-vector division per row or column to be zeroed. The division found in the Householder reflection may be easily reformulated as a scalar-vector product. Since the division and square root operations are very expensive to compute, the Householder reflection is preferred over the

Givens rotation. An example of a Householder-based QR decomposition is given in Figure 5.15.

```
#define N 5
void QRHouse (float fA[N][N])
{ float fV[N], fT[N], fMu, fBeta;
  int iJ, iN, iM;

  for (iJ=0; iJ<N-1; ++iJ) {
    fV[0:N-iJ-1]=fA[iJ:N-1][iJ]; /* Extract column to zero. */

    fT[0:N-iJ-1]=fV[0:N-iJ-1]*fV[0:N-iJ-1]; /* Calc. 2-norm of fV. */
    fMu=sqrt(1.0 $$ fT[0:N-iJ-1]);

    if (fMu>0.0) { /* Compute Householder vector. */
      if (fV[0]>0.0) fBeta=1.0/(fV[0]+fMu);
      else fBeta=1.0/(fV[0]-fMu);
      fV[1:N-iJ-1]=fBeta*fV[1:N-iJ-1];
    }
    fV[0]=1.0;

    /* Apply Householder vector to original matrix. */
    fBeta=-2.0/(fV[0:N-iJ-1] $$ fV[0:N-iJ-1]); /* Scale factor. */
    for (iN=0; iN<N-iJ-1; ++iN) /* Update vector. */
      fT[iN]=fBeta*(fA[iJ:N-1][iJ+iN] $$ fV[0:N-iJ-1]);
    for (iN=0; iN<N-iJ-1; ++iN) /* Apply outer product update. */
      for (iM=0; iM<N-iJ-1; ++iN)
        fA[iJ+iM][iJ+iN]+=fV[iN]*fT[iM];

    fA[iJ+1:N-1][iJ]=fV[1:N-iJ-1]; /* Save Householder vector. */
  }
}
```

Figure 5.15: C_{DSP} Function for QR Decomposition

The QR decomposition implementation shown in Figure 5.15 is designed to compute the QR decomposition of a square matrix of fixed size. The underlying algorithm is derived from algorithms presented in Golub and Van Loan [5]. It should be noted

that the self-contained C_{DSP} source is substantially more compact than a comparable self-contained C implementation. Not only is the C_{DSP} implementation compact, but many of the C_{DSP} semantic extensions to the C language used in the function map directly into efficient DSP microprocessor code.

Opportunities for block level parallelism can be explored by breaking the algorithm down into its three main algorithmic components, which are

- computation of the Householder vector,
- computation of the update vector, and
- computation of the outer product update to the original matrix.

These components are applied to iteratively to successively smaller sub-matrices of the original matrix to produce the QR decomposition. Within each iteration of the outer-most loop, it can be seen that the computation of the update vector is dependent upon computation of the Householder vector, and that the computation of the outer product update is dependent upon the computation of the update vector. Due to the overlapping outer product updates between iterations of the outer-most loop, it is not possible to parallelize the individual iterations of the loop. However, by examining the computation of the update vector and the computation of the outer product update it can be seen that there is an opportunity for parallelism between these computations. The impact that can be achieved by exploiting this opportunity for block level parallelism is illustrated in Figure 5.16.

5.3.4 Results

The algorithms explored in this section are among the cornerstones of digital signal processing. The algorithms were demonstrated to have opportunities for parallelism that could be exploited by a VLIW DSP processor. Exploiting the opportunities

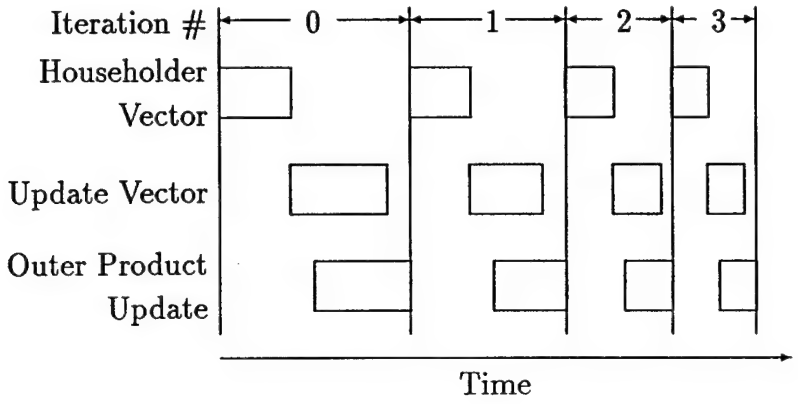


Figure 5.16: Diagram of Execution Timing and Exploitable Block Level Parallelism for Householder QR Decomposition

for parallelism illustrated in Sections 5.3.1, 5.3.2, and 5.3.3 does not require special effort on the part of the programmer in the VLIW/C$_{DSP}$ programming environment described herein. Methods of performing dependency analysis and performing instruction scheduling are well-known [47, 48]. The significance of these results are that VLIW architecture allows relatively inexpensive parallel computing while the defined C$_{DSP}$ language allows rapid, efficient implementation of parallel processing software.

CHAPTER 6
CONCLUSIONS

6.1 Summary

Digital signal processing applications frequently demand high arithmetic bandwidth; small, inexpensive packaging; low power consumption and dissipation; and low cost. These attributes are generally interrelated and at odds with one another. The implementation technology of choice for digital signal processing applications is the DSP microprocessor. Semiconductor technology has progressed to the point where multiprocessing DSP solutions can be constructed, however, existing solutions have been less than satisfactory. Very long instruction word architectural techniques have great promise for enabling high-speed DSP multiprocessing with superior power, packaging, and cost factors compared to existing DSP multiprocessing solutions.

Chapter 2 introduces the residue number system and describes the existing state of the RNS theory. Chapter 3 describes the Athena Sensor Arithmetic Processor, an application specific SIMD digital signal processor that uses the RNS. The ASAP device achieves peak performance of 1.2 billion thirty-two bit multiply-accumulate operations per second using less than 20 mm^2 of die area when fabricated in the MOSIS 0.8 μm CMOS process. The ASAP device demonstrates a one to two order of magnitude speed-area advantage over conventional arithmetic implementations, depending upon the computation performed. The ASAP technology provides motivation for pursuing a VLIW DSP microprocessor architecture in that it makes an ideal functional unit for such a processor. In turn, the VLIW DSP microprocessor

offers balance not found in previously reported RNS implementations in the form of conventional arithmetic units that are able to perform those operations for which the RNS is poorly suited.

In Chapter 4 the architectural elements required for VLIW digital signal processing are explored. The architectural elements include conventional arithmetic units, RNS arithmetic units and supporting functional units, local memories connected to functional units by a switch, and a DMA controller to handle transfers between on-chip and off-chip memories. The architecture described is a block load-store architecture with block load and store operations performed by the DMA controller under programmed direction. Providing global switched access to the local memory resources limits the architecture due to the geometric growth of the expense of the switching elements with respect to the number of functional units and local memories. This leads to the conclusion that a hierarchical non-uniform memory access model is required to linearize the resources consumed by the switching elements versus the number of functional units and local memories.

VLIW digital signal processing presents a substantial problem, namely programming the VLIW DSP microprocessor. Writing VLIW machine code directly is similar to writing horizontal microcode: the smallest functions require Herculean programming efforts. Even a micro-instruction oriented assembler with automatic instruction scheduling would present a difficult working environment to the application programmer. Furthermore, in either of these models, porting applications to architectural variants of the same processor would require substantial re-engineering of the application.

To address the problems associated with programming a VLIW DSP processor, a high-level assembly language, C_{DSP} , has been defined. The C_{DSP} language is optimized for DSP applications that will be executed on VLIW DSP microproces-

sors. The C_{DSP} language provides excellent programmer productivity and code portability is aided by the ability to retarget the application to a new processor by recompilation. Not only is the C_{DSP} language optimized for DSP applications, it is also optimized for automatic parallelization of DSP applications, particularly for VLIW DSP processor architectures. A significant benefit realized by using a high-level language for parallel DSP compilation is the additional information that is available to the compiler compared to that available to an assembler.

In Chapter 5, the impact of the C_{DSP} langauge and parallelism in the VLIW DSP environment were examined in the context of three cornerstone DSP algorithms: convolution and FIR filtering, discrete Fourier transforms, and the QR decomposition. In all three cases C_{DSP} implementations were shown and opportunities for parallelism within these implementations were demonstrated.

6.2 Contributions

The main contibutions made in this dissertation were:

- Demonstrated an LRNS processor capable of up to 1.2 billion operations per second, with a one to two order of magnitude speed-area advantage over processors fabricated using conventional technologies.

- Developed a high-level programming language, C_{DSP} , for VLIW DSP. The C_{DSP} language is highly optimized both for digital signal processing applications and for parallel computing, a synergism that makes it ideal for VLIW DSP.

- The C_{DSP} language enables selection of a processor to just fit an application by allowing the application to be written before the target hardware is selected.

- The practical limits of N-way VLIW have been explored. The conclusion was that full, globally switched interconnect between functional units for large N is impractical (expensive) and undesirable (not needed by likely applications).

- To scale VLIW to very large numbers of functional units, a three level NUMA processor-local memory switch architecture has been designed. This architecture allows individual, unrelated threads of execution to be executed on separate processor groups without causing contention for global switch resources.

6.3 Future Work

There are a number of problems that remain to be solved in the area of VLIW DSP processors. The analysis presented in this dissertation was based upon a high-level description of a VLIW DSP microprocessor. Since this research began, at least one two-way VLIW DSP microprocessor has become available as a standard part (Texas Instruments' C6200). While this is significant, it is still quite far from a large N-way VLIW DSP microprocessor. With the explosive growth of ASIC implementation methodologies, there is clearly a potential need for a customizable VLIW DSP microprocessor core for ASICs. Whether that core is hard or is synthsizable it is clearly desirable to be able to specify no more microprocessor than is required to solve the problem at hand.

The RNS functional units described in this dissertation demonstrate that an application accelerator can have great value in a microprocessor environment. In an ASIC environment where the population of the processor's functional units is configurable by the user, the development of more application specific accelerator functional units is clearly desirable. For example, a VLIW DSP processor that is intended for video processing applications would greatly benefit from an 8×8 discrete cosine transform accelerator.

The current status of the C_{DSP} compiler is that it is a compiler front-end, implemented with standard compiler construction tools (YACC, LEX, etc.). Completion of C_{DSP} compiler and targeting of the compiler at configurable VLIW DSP microprocessor would allow more quantitative architectural studies to be performed. Since the C_{DSP} language is optimized for DSP microprocessors, it would also be valuable to target the compiler to standard DSP microprocessors and quantitate its performance versus C compilers and assembly language for those standard processors. Ultimately, given the weight of experience the C_{DSP} language should be revised to correct any significant oversights and to add capabilities that would benefit unforeseen architectural feature and applications.

APPENDIX A
C_{DSP} LANGUAGE REFERENCE

A.1 Introduction

This document is the language reference manual for the C_{DSP} programming language for digital signal processors. The C_{DSP} language is based upon the principals of the C programming language; principally that the C_{DSP} language is a high-level assembly language for digital signal processors. In particular, this manual is derived from the ANSI C standard [49]. Why not just use C? The original C language was, in fact, a high level assembly language for the DEC PDP-11 [50] and its successors. In fact, many of the PDP-11's successors, particularly the RISC microprocessors that dominate the desktop workstation market, are designed so that C is an effective high-level assembly language. Digital signal processors are not designed to allow C compilers to generate optimal code for signal processing. In fact, digital signal processors that are optimum hardware for running signal processing algorithms can only be supported in a marginal sense by C compilers — usually via hand coded assembly language libraries and idiomatic translation.

A.2 Notation

This manual formally defines a grammar for the C_{DSP} language using a series of rules, or productions. The format used for these productions is a modified Backus-Naur form (BNF).

Terminals in the grammar are denoted using a monospaced font, for example, "`if`." Non-terminals are denoted using italics, for example, "*expression.*" Optional

terminals and non-terminals are enclosed in square brackets, for example, "[*optional*]." When a choice between more than one terminal or non-terminal is required those choices are separated by a vertical bar (|), for example, "*choice1*|*choice2*." A non-terminal that is used in a production outside of the section where it is defined will be tagged with its defining production number. Subsequent references to that non-terminal will not, however, be tagged with a reference to the defining production. The non-terminal defined by a production appears to the left of the symbol "::=" while the matching rules of the production appear to the right.

Regular expressions that are used to match terminals are defined using the usual Unix regular expression syntax.

A.3 Lexical Elements

A.3.1 Character set

The only characters used in the C_{DSP} language are defined in Table A.1. All defined characters are members of the ISO seven-bit standard character set (ISO 646-1983) and their representations in this manual are the ASCII defined representations.

Table A.1: The C_{DSP} Character Set

(1) alphabetic characters
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z
(2) digits
0 1 2 3 4 5 6 7 8 9
(3) special characters
[ ] ( ) = + − * / { } , .
(4) space
(ANSI space character)

Source programs may be contain any of the characters defined in Table A.1 plus the usual whitespace formatting characters: horizontal tab, vertical tab, carriage return, line feed, and form feed.

A.3.2 Abstract literals

There are three defined classes of abstract literals: integers, reals, and strings. These abstract literals are defined in the following discussion

Integer literals. An integer literal may take several forms. In particular, an integer literal may be expressed in base ten (decimal), base eight (octal), or base sixteen (hexadecimal). Integral values may also be specified with a character literal.

A decimal literal may begin with a unary negation (-) to indicate that the number is negative. After the unary negation the literal may only contain the digits zero through nine and may only start with zero if the literal is identically zero. An octal literal is always interpreted as an unsigned value. The octal literal must begin with the digit zero and is subsequently followed by one or more digits in the range of zero through seven. Like the octal literal, the hexadecimal literal must always be interpreted as an unsigned value. The hexadecimal literal must always begin with either of "`0x`" or "`0X`" and must be followed with one or more digits in the range of zero through nine or alphabetic characters in the range "a" through "f," whether in upper or lower case. Finally, an integral literal may be expressed using a character literal. A character literal is a "single" character enclosed in single quotation marks. Note that the single character may be an escape sequence that begins with a backslash. These literals and the regular expressions that match them are given in Table A.2.

Table A.2: Regular Expressions for Integral Literals

Literal Type	Regular Expression
octal	`0[0-7]+`
decimal	`-?[0-9]+`
hexadecimal	`0(x|X)[0-9A-Fa-f]+`
character	`'\\?.'`

Escape sequences. It is desirable to allow any character to be represented in a string or character literal. Within the allowable character set for source programs (see Section A.3.1) it is not possible to directly place any character in a character or string literal. In fact, few terminals will even allow a programmer to enter any possible character. Therefore it is necessary to provide a mechanism that allows the programmer to enter "special" characters in character and string literals. The mechanism to allow this to be done is called an *escape sequence.* Escape sequences may only be found in character and string literals and always begin with a backslash followed by one or more characters that may have significance.

Table A.3: Escape Sequences for Character and String Literals

Escape Sequence	Description
\n	Newline.
\t	Horizontal tab.
\r	Carriage return.
\v	Vertical tab.
\a	Alert or bell.
\f	Form feed.
\0YYY	A character with the octal value YYY. From one to three octal digits must follow the leading zero.
\xYY	A character with hex value YY. One or two hex digits must follow the "x."
\'	Single quote.
\"	Double quote.
\\	Backslash.
\0	Null character

Floating-point literals. Floating-point literals may assume the usual forms, namely

1. Fixed-point format (e.g., "`3.5`"). At least one digit must occur both before and after the radix point.

2. Scientific notation format (e.g., "`3.5e2`"). The previous rules for the fixed-point format govern the mantissa portion of this literal format. The letter "`e`" may be either uppercase or lower case. After the letter "`e`" there may be a unary plus or minus, but neither is required. Finally, a decimal integer follows. The usual interpretation of this format applies (i.e., `3.5e2`$= 3.5 \times 10^2$).

The regular expression to match a floating-point literal in either of the above cases is given in Table A.4.

Table A.4: Regular Expression for Floating-Point and Fixed-Point Formats

Literal Type	Regular Expression
floating-point	`[0-9]+\.[0-9]+([eE][-+]?[0-9]+)?`

String literals. String literals in C_{DSP} take the usual form found in the C language. The rules for the formation of a string literal follow.

1. String literals are delimited with a double quote ("`"`").

2. String literals may contain the direct representation of any ANSI alphabetic, numeric, or punctuation character.

3. String literals may not directly span a newline. In order to span a newline a string literal must be closed with a "`"`" and restarted with another "`"`". The only intervening symbols allowed in the source file are whitespace characters.

String literals may contain the escape sequences defined in Table A.3.

A.3.3 Comments

Comments in C_{DSP} use the ANSI C form for comments. Comments may begin at any point (except in a string or character literal) and end at any point. Comments start with the two character sequence "`/*`" and end with the two character sequence

"*/". Comments may span multiple lines in the source file. The C++ single line comment "//" is *not* supported. Comments do not in any way effect the code generation or execution of code in a C_{DSP} program.

A.3.4 Identifiers

Identifiers are used to name storage objects, functions, labels, and user defined types. Identifiers in the C_{DSP} language must conform to the following rules:

1. Identifiers may be composed of uppercase and lowercase alphabetic characters, digits, and the underscore character.

2. Identifiers may not begin with a digit.

3. All reserved words are identifiers and may not be used in an explicit declaration.

4. The interpretation of identifiers is case sensitive (e.g., "if" is not the same as "If").

5. Only the first thirty-one characters of an identifier are required to be considered significant within a translation unit (see Section A.4). Implementations may elect to consider more characters. When linking translation units the number of significant characters is implementation defined.

Identifiers generally have limited visibility or scope. An identifier is never in scope until after it is declared or defined. Function identifiers are in scope from the initial point of declaration (prototype) or definition to the end of the translation unit.

Storage elements and user defined types that are declared or defined outside of the body of functions are also in scope to the end of the translation unit. Storage objects that are declared outside the body of functions is said to be global. A storage object that is declared or defined within the body of a function or a compound statement is

in scope only to the end of the containing function or compound statement, however, type definitions defined within a function have global scope. A storage object that is declared or defined in the body of a function or compound statement is said to be local. The identifier for a global storage object or type definition may not be redefined with global scope, however, a local storage object may be defined using the same identifier as that used for a global or in a containing local scope. If the identifier for a storage object is redefined within a local scope, the newly defined storage object takes precedence over that associated with the containing scope.

Labels only have scope within the function and are in scope for the entire function.

A.3.5 Reserved words

The C_{DSP} reserved words are listed in Table A.5. All reserved words are lowercase and permutations on the case of the reserved words will not be matched as reserved words (see Section A.3.4).

Table A.5: C_{DSP} Reserved Words

auto	break	char	const	continue
do	dopar	else	extern	fixed
float	for	if	index	int
long	return	short	signed	static
unsigned	void	volatile	while	

A.4 Translation Unit

A C_{DSP} source file is known as a *translation unit.* A translation unit may be either empty of contain declarations and definitions. Declarations and definitions may be given for both functions and storage (variables, constants). The productions that define a translation unit are given below.

(A.1) *file* ::= ϵ | *translation_unit*

(A.2) *translation_unit* ::= *external_declaration* |

translation_unit external_declaration

(A.3) *external_declaration* ::= *function_definition*(A.4) |

declaration(A.27)

A.4.1 Function definitions

The production for a function definition is given below. In Kernighan and Ritchie C [51] (sometimes referred to as "K&R C"), a *declaration_list*(A.51) was placed between the *declarator* and the *compound_statement* to allow the types of the elements parameter list to be defined. ANSI C preserved this as an "old style" function header to support migration of legacy Kernighan and Ritchie-style code, however, that option does not exist in C_{DSP} since there is no legacy C_{DSP} code to support.

(A.4) *function_definition* ::= [*declaration_specifiers*(A.28)] *declarator*(A.35)

compound_statement(A.50)

A function definition must contain a *declarator* and a *compound_statement.* If *declaration_specifiers* are not explicitly given then the function definition has a default type of `int`.

C_{DSP} functions do not use a stack for storage. All local storage uses statically determined locations. There are many advantages to this approach. First, by using statically determined storage locations code generation is simplified since dynamic storage does not have to be managed. Furthermore, in a multi-threaded execution environment, the stack-based memory allocation method preferred for automatic storage is difficult to implement, particularly since many DSP microprocessors lack a stack for data storage. Using statically determined storage locations for storage also eliminates the storage linkage operations usually performed upon entering and exiting any block where local storage is allocated.

While all functions in the C programming language are recursive, functions in the C_{DSP} programming language are *not* recursive. Recursion is easy to support when a stack model is used for automatic storage (`auto`), however, since the C_{DSP} execution environment does not use a stack, supporting recursion would be difficult. Furthermore, the dynamic memory usage requirements of recursive functions cannot be predicted and are, therefore, impossible to schedule at compilation time. Recursion is a very useful programming technique for some applications, such as transversing a tree structure, however, DSP applications do not usually have many complicated data structures. Recursion can also be used for numerical computations, however, such implementations of numerical computations are often grossly inefficient, expending the majority of their execution time in the function call and return processes. In either event, recursion can always be simulated.

C_{DSP} functions are not reentrant. Reentrancy, like recursion, presents special implementation challenges. For instance, any `static` storage associated with a function must be managed with a mutually exclusive (MUTEX) lock. This is particularly difficult as hardware support for a MUTEX lock cannot be assumed in a multiple DSP microprocessor execution environment. Automatically allocated storage must also be separately managed, presenting a difficult dynamic memory allocation problem.

A.4.2 External object definitions

Global storage objects and function objects may be defined in external translation units. Access to externally defined objects is mediated by a linker. The operation of the linker is not defined in this document. A global object (storage or function) that is defined with the `static` storage class are not in scope outside of the translation unit where it is defined.

A.5 Conversions

The C_{DSP} language, like its namesake, is loosely typed. That is, expressions involving operands of mixed type are allowed. In order to support operations involving operands of mixed type it is necessary to automatically convert operands of mixed type to a common type so that the operation can be performed. Operations where operands are automatically converted to compatible types are said to employ "automatic type conversion." The implicit conversion of operands discussed here is contrasted with the explicit conversion accomplished using cast operators (see Section A.6.4).

Automatic type conversion in the C_{DSP} language is value preserving. In other words, when an automatic type conversion is to be performed, the resulting type will capable of representing the value to be converted. Table A.6 shows the direction of automatic type conversion of intrinsic scalars in the C_{DSP} language; automatic type conversions will only convert a value to a type that is lower in the list in Table A.6. Conversions of signed values are sign-preserving (i.e., sign-extension is performed).

Table A.6: Direction of Automatic Type Conversions

`char`	least precedence
`unsigned char`	↓
`short`	↓
`unsigned short`	↓
`int`	↓
`unsigned int`	↓
`long`	↓
`unsigned long`	↓
`fixed`	↓
`float`	greatest precedence

The `fixed` type may have up to four sizes `char` (eight bits), `short` (sixteen bits), `int` (between sixteen and thirty-two bits), and `long` (thirty-two bits). Automatic type conversion of an integral type versus a `fixed` will result in a `fixed` representation

of the same size as the integral type if the `fixed` value is smaller than the integral value. For example, a `fixed(char)` multiplied by a `short` will produce a result of type `fixed(short)`. Type resolution and automatic type conversion of operands of differing `fixed` sizes will likewise result in a `fixed` size equal to the larger of operands.

Conversions of integral and `fixed` values to floating-point values will result in a floating-point value that is as close as possible for the given implementation.

Forced conversions (casts) from the floating-point representation to a `fixed` representation will map the floating-point value according to the parameters of the `fixed` representation. Mapping of floating-point values to integral representation results in the truncation of all fractional bits in a normalized representation. If the floating-point value is too large to be represented given the chosen `fixed` or integral type then the result is an undefined value. Conversions from `fixed` representations to integral representations results in the truncation of all fractional bits. The conversion of a `fixed` to integral type proceeds as if the `fixed` value were first converted to an integral version of its "container" type (e.g., `fixed(char)` to `char`) and then an integral to integral conversion is performed, if necessary.

Integral conversions from larger to smaller types are performed by truncating the high-order bits of the larger type. If the original value were in the range of the target type then this conversion will be value preserving. However, if the original value is not in the range of the target type then the resulting value will be undefined.

A.6 Expressions

A.6.1 Primary expressions

A primary expression is either a constant, a string literal, a parenthesized expression (Production (A.25)), or an identifier. An identifier may be a primary expression if and only if it has been previously declared or defined as a variable, constant, or

function.

(A.5) *primary_expression* ::= *identifier* |

constant |

string_literal |

(*expression*(A.25))

A.6.2 Postfix operators

A postfix expression followed by a set of square brackets "[]" enclosing an expression (Production (A.25)) is an array element reference (i.e., a subscript). The subscripting expression must have a scalar integral value. A postfix expression followed by a parenthesized argument expression list (possibly empty) is a function reference providing the function has been previously defined or declared.

Sub-arrays may also be specified using colon notation. In the first case a range of indexes may be specified using the notation

start:*stop*

where *start* and *stop* are expressions and *stop* is greater than or equal to *start*. This notation specifies all indexes from *start* to *stop*, inclusive. Sub-arrays with non-unit index stride may be specified with the notation

start:*stop*:*stride*

where *start* and *stop* are non-negative expressions, and *stride* is a non-zero expression. If *stride* is positive then *stop* should be greater than or equal to *start*, while if *stride* is negative then *stop* should be less than or equal to *start*. If A is the starting index, B is the stopping index, and S is the stride, then let

$$L = \left\lfloor \frac{|B - A|}{|S|} \right\rfloor . \tag{A.1}$$

Given L, then the ordered set of indexes specified by the notation $A:B:S$ is

$$\{A, A+S, A+2S, \ldots, A+(L-1)S\}. \tag{A.2}$$

From this it is clear that the set of indexes does not pass B.

The application of index range notation to arrays is equivalent to the formation of a new array with elements selected from the original array according to Equation A.2, producing an ordered mapping to the index set

$$\{0, 1, 2, \ldots, L-1\}. \tag{A.3}$$

The post-increment "`++`" and post-decrement "`--`" operators operate only on scalar integral types. They operate in the usual way (incrementing or decrementing by one) when operating on any scalar type except the `index` type. The operation of the increment and decrement operators upon the `index` type is controlled by the attributes supplied with the dot operator discussed below.

(A.6) *postfix_expression* ::= *primary_expression* |
postfix_expression`[`*expression*(A.25)`]` |
postfix_expression`[`*expression*`:` *expression*`]` |
postfix_expression`[`*expression*`:` *expression*`:` *expression*`]` |
postfix_expression`(`[*argument_expression_list*]`)` |
postfix_expression`.mod` |
postfix_expression`.stride` |
postfix_expression`.bitrev` |
postfix_expression`.base` |
postfix_expression`.ind` |

postfix_expression++ |

postfix_expression--

(A.7) *argument_expression_list* ::= *assignment_expression*(A.23) |

argument_expression_list, *assignment_expression*(A.23)

The rules for *postfix_expression*.{`mod` | `stride` | `bitrev` | `base` | `ind`} are provided for the `index` variable type which, under normal operation has state information besides the current value of the index. The "." operator is used in the C language to support member access of `structs`. Since `structs` do not exist in the C_{DSP} language, the "." operator has been appropriated to identify `index` attributes.

The various attributes of a variable of type `index` are illustrated in Figure A.1. The `.ind` attribute is the basic index value of the `index` type and is a signed integral scalar. The `.ind` attribute is the one that is changed when the increment and decrement operators are applied to the `index` (i.e., `iN`, not `iN.ind`). The `.mod` attribute controls the modulus used with increment and decrement operations on the `index` and is an unsigned integral scalar. If the value of the the `.mod` attribute is zero then no modulus operation is performed when incrementing or decrementing the index. The `.stride` attribute is a signed integral scalar that is the value that is added to the index when it is incremented (or subtracted when it is decremented). If the `.mod` attribute is non-zero then the absolute value of the `.stride` attribute should be less than the `.mod` attribute for a particular `index` value, otherwise the effects of an increment or decrement operation are undefined. The `.base` attribute of an `index` value is an offset, allowing the use of modular addressing within a sub-array of a larger array.

The semantics of the increment and decrement operators and the impact of the attributes of an `index iN` are given as follows.

1. The value of `iN` is a signed integral scalar equal to `iN.base+iN.ind`.

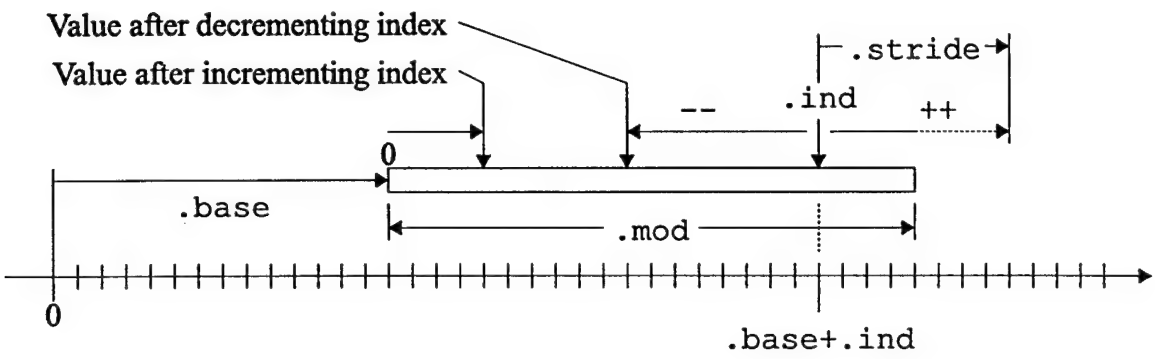


Figure A.1: Semantics of index Attributes

2. If iN.mod is zero then the increment (decrement) operator applied to iN (either ++iN or iN++) results in iN.ind←iN.ind+(−)iN.stride.

3. If iN.mod is greater than zero then the increment (decrement) operator applied to iN results in iN.ind←(iN.ind+(-)iN.stride) mod iN.mod.

The .bitrev attribute is an integral scalar. If the value of .bitrev is zero then the semantics of the index value under increment and decrement operators are as given above. If .bitrev is non-zero and .mod is non-zero then the semantics of the index value under the increment and decrement operators is undefined. If .bitrev is non-zero and .mod is zero then the semantics of the increment operator are to cause the following action: iN.ind←iN.ind$\vec{+}$iN.stride, where $\vec{+}$ indicates binary addition with reversed carry propagation. For example, $110_2\vec{+}100_2 = 001_2$. If .bitrev is non-zero and .mod is zero then the semantics of the decrement operator are undefined.

A.6.3 Unary operators

The pre-increment and pre-decrement operators operate only on scalar integral types. As previously discussed, these operators work in the usual way with integral scalar types, and with the special semantics described in the previous section for values of type index.

(A.8) *unary_expression* ::= *postfix_expression* |

++*unary_expression* |

--*unary_expression* |

unary_operator cast_expression(A.10) |

`sizeof` *unary_expression* |

`sizeof` (*type_name*)

(A.9) *unary_operator* ::= & | + | - | ~ | !

The `sizeof` operator produces a constant (since it is evaluated at compile time) unsigned integral scalar and may be used to evaluate a type either by referencing the type name or an expression.

The unary operators are the "address-of" operator (&), the unary plus (+), the unary minus (-), the bitwise NOT (~), and the logical NOT (!). The unary plus, minus, bitwise NOT and logical NOT operators operate upon scalar and array values. The unary plus and minus operations work in the usual way. The bitwise NOT operation causes the negation of each bit of the operand.

The logical NOT operation produces a result of zero (false) if the value of the operand is non-zero (true) and one (true) if the value of the operand is zero (false). The type of the result of the logical NOT operation is always `int`, regardless of the type of the operand.

A.6.4 Cast operators

A cast may be applied to an expression by placing a valid type name in parenthesis in front of the expression to be converted. A conversion of an expression to a "larger" type will preserve the value of the expression converted. A conversion to a "smaller" type (e.g., `int` to `char`) will preserve the value of the expression converted if the value of the original expression is in the range of the new type, otherwise the value

resulting from the conversion is undefined. A cast may be applied to both scalar and array expressions.

(A.10) *cast_expression* ::= *unary_expression*(A.8) |

(*type_name*(A.42)) *cast_expression*

A.6.5 Convolution and sum of products operators

The convolution operators are linear convolution ($) and circular convolution (@). These operators operate on array operands. If either operand is a scalar then the operation is reduced (and equivalent to) a scalar multiplication.

(A.11) *convolution_expression* ::= *cast_expression*(A.10) |

convolution_expression $ *cast_expression* |

convolution_expression @ *cast_expression*

convolution_expression $$ *cast_expression*

Linear convolution operands must have the same dimensionality, unless one operand is a scalar, in which case the operation is interpreted as a multiplication. The linear convolution is computed in the usual way, with the size of the result in each dimension equal to sum of the operand sizes in that dimension minus one.

Circular convolution may be performed using array operands of differing sizes, however, the operands must have the same dimensionality. The circular convolution will be computed as if the operand with the smaller size in a particular dimension is zero padded in that dimension to match the size of operand with the larger size in that dimension. For instance, the circular convolution of a 3×2 array with a 2×3 array will cause the computation to proceed as if each had been zero padded to 3×3 elements, producing a 3×3 result.

The $$ operator is the sum of products operator. The sum of products operator is an array operator. The sum of products operands must have the same geometry,

unless one operand is a scalar, in which case it will be taken as an array with the same geometry as the array operand and each element of the array has the scalar's value.

The means used to perform convolution and sum of products computations are not specified and are implementation dependent.

A.6.6 Multiplicative operators

The multiplicative operations are multiplication (*), division (/), and the remainder or modulus operation (%). All three of these operators operate both upon scalars and arrays. If both operands are array operands then the arrays must have identical geometry. If one operand is a scalar and the other is an array then the computation will be performed as if the scalar operand were actually an array with the same geometry as the actual array operand with all elements having the same value as the scalar operand. Before the operation is performed, if one of the operands is of a smaller type than the other then it will be converted to the larger type before the operation is performed, with the result taking the larger operand type.

(A.12) *multiplicative_expression* ::= *convolution_expression*(A.11) |
multiplicative_expression * *convolution_expression* |
multiplicative_expression / *convolution_expression* |
multiplicative_expression % *convolution_expression*

If the results of a multiplication operation overflow the capacity of the type used for the multiplication and undefined result is produced. Multiplication of arrays proceeds element by element rather than using the matrix multiplication algorithm.

The division operation is undefined if the second operand has a value of zero. When division is performed using integral operands of the same sign the quotient will

be truncated towards zero,

$$x/y = q + r, \tag{A.4}$$

where q is an integer and $r \in [0, 1)$. When the signs of the operands are different the direction of truncation (towards zero or away from zero) are implementation dependent. That is, if sign(x)≠sign(y), then q is an integer as before and either $r \in [0, 1)$ *or* $r \in (-1, 0]$.

The modulus or remainder operation is only defined over the integral types and only if the second operand is non-zero. The value produced by the modulus operation is defined by the relationship **(x/y)*y+x%y** is equal to **x**. As a result, the value produced when one of the operands has a negative value will depend upon the quotient produced by the division operation and is, therefore, implementation dependent.

A.6.7 Additive operators

The rules for the automatic type conversion of operands of the additive operators are given in Section A.6.6. The addition and subtraction operations work in the usual way. If the results of an additive operation overflow the capacity of the type used to perform the expression then the result is undefined. If one or both operands are arrays then the addition or subtraction operation will be performed element-by-element, as described in Section A.6.6.

(A.13) *additive_expression* ::= *multiplicative_expression*(A.12) |
additive_expression + *multiplicative_expression* |
additive_expression - *multiplicative_expression*

A.6.8 Bitwise shift operators

The shift operators are used to perform logical shifts of integral values. The result of either shift operation is undefined if either of the operands is not integral, or if the

second operand is negative or greater than the width (in bits) of the first operand minus one. The type of the result will be the same as the type of the first operand; the type of the second operand does not impact the type of the result.

(A.14) *shift_expression* ::= *additive_expression*(A.13) |

shift_expression << *additive_expression* |

shift_expression >> *additive_expression*

The left shift operation (<<) shifts the first operand left the number of bits specified by the second operand. The right shift operation (>>) shifts the first operand right the number of bits specified by the second operand. In the case of the left shift, the number given by the second operand of the least significant bits is set to zero, while in the case of the right shift, the number given by the second operand of the most significant bits is set to zero. In other words, in both cases zeros are shifted into the new value. The bits shifted out are not preserved. Furthermore, these shifts are *not* arithmetic (sign preserving).

If one or both operands are arrays then the operation will be performed element-wise as described in Section A.6.6.

A.6.9 Relational operators

The relational operators, less-than (<), greater-than (>), less-than-or-equal (<=), and greater-than-or-equal (>=), take scalar operands and produce `int` results. The value of the expression will be zero if the relational operation evaluates to be false, and one if the relational operation evaluates to be true. As in Section A.6.6, the operands will be automatically converted to compatible types before the comparison occurs.

(A.15) *relational_expression* ::= *shift_expression*(A.14) |

relational_expression < *shift_expression* |

relational_expression > *shift_expression* |

relational_expression <= *shift_expression* |

relational_expression >= *shift_expression*

A.6.10 Equality operators

The equality operators, equality (==) and inequality (!=), take scalar operands and produce `int` results. The value of the expression will be zero if the operation evaluates to be false and one if the expression evaluates to be true. As in Section A.6.6, the operands will be automatically converted to compatible types before the comparison occurs.

(A.16) *equality_expression* ::= *relational_expression*(A.15) |

equality_expression == *relational_expression* |

equality_expression != *relational_expression*

Note that all comparisons are *exact*, therefore, these operations probably have limited utility when non-integral types are used.

A.6.11 Bitwise AND operator

The bitwise AND operation is performed on each bit of the operands. The bitwise AND operation is only defined if both operands are integral. As in Section A.6.6, the operands will be automatically converted to compatible types before the operation occurs.

(A.17) *AND_expression* ::= *equality_expression*(A.16) |

AND_expression & *equality_expression*

If one or both operands are arrays then the operation will be performed element-by-element as described in Section A.6.6.

A.6.12 Bitwise exclusive OR operator

The bitwise exclusive OR operation is performed on each bit of the operands. The bitwise exclusive OR operation is only defined if both operands are integral. As in Section A.6.6, the operands will be automatically converted to compatible types before the operation occurs.

(A.18) *exclusive_OR_expression* ::= *AND_expression*(A.17) |
exclusive_OR_expression ^ *AND_expression*

If one or both operands are arrays then the operation will be performed element-by-element as described in Section A.6.6.

A.6.13 Bitwise inclusive OR operator

The bitwise inclusive OR operation is performed on each bit of the operands. The bitwise inclusive OR operation is only defined if both operands are integral. As in Section A.6.6, the operands will be automatically converted to compatible types before the operation occurs.

(A.19) *inclusive_OR_expression* ::= *exclusive_OR_expression*(A.18) |
inclusive_OR_expression | *exclusive_OR_expression*

If one or both operands are arrays then the operation will be performed element-by-element as described in Section A.6.6.

A.6.14 Logical AND operator

The logical AND operation will produce zero (false) if one or the other operand (or both operands) is zero (false), and will produce one (true) if both of the operands are non-zero (true). If the first operand is zero (false) then the second operand will not be evaluated. The logical AND operation is only defined for scalar operands. The result of the logical AND operation is a value of type `int`.

(A.20) *logical_AND_expression* ::= *inclusive_OR_expression*(A.19) |

logical_AND_expression && *inclusive_OR_expression*

A.6.15 Logical OR operator

The logical OR operation will produce zero (false) if both operands are zero (false), and will produce one (true) otherwise. If the first operand is non-zero (true) then the second operand will not be evaluated. The logical OR operation is only defined for scalar operands. The result of the logical OR operation is a value of type `int`.

(A.21) *logical_OR_expression* ::= *logical_AND_expression*(A.20) |

logical_OR_expression || *logical_AND_expression*

A.6.16 Conditional operator

The conditional operation is performed using a ternary operator. The operands must be scalars. If the first operand is non-zero (true) then the value of the operation is given by the second operand-expression, while if the first operand is zero (false) then the value of the operation is given by the third operand-expression. The operand-expression that is not selected is not evaluated.

(A.22) *conditional_expression* ::= *logical_OR_expression*(A.21) |

logical_OR_expression ? *expression* : *logical_AND_expression*

A.6.17 Assignment operators

The assignment operator (=) is a right associative operator that takes any expression as its second operand (usually called the *rvalue* for right-hand side) and assigns it to the location specified by the first operand (usually called the *lvalue* for left-hand side). The *rvalue* may be any expression, however, the *lvalue* must specify a valid storage location. The *lvalue* and *rvalue* must both be scalars. If the type of the *rvalue* is of a smaller type than the *lvalue* then it will be automatically converted to

the type of the *lvalue*, however, in the case of the opposite conditions, the value of the assignment is undefined. All assignment operations also produce a value, namely the value assigned to the *lvalue*.

(A.23) *assignment_expression* ::= *conditional_expression*(A.22) |
unary_expression assignment_operator assignment_expression

(A.24) *assignment_operator* ::= = | *= | /= | %= | += | -= | <<= | >>= | &= | ^= | |=

The remaining assignment operators are referred to as compound assignment operators because they result in an operation and an assignment. Each compound assignment operator has an equivalent expression using other operators; the previously stated restrictions regarding the type of the *lvalue* and *rvalue* hold for the compound assignment operations, as well as additional restrictions that are the restrictions on the original operations. The compound assignments and their equivalents are summarized in Table A.7.

Table A.7: Compound Assignment Operations and Equivalent Assignments

Compound Assignment	Equivalent Assignment
x*=y	x=x*y
x/=y	x=x/y
x%=y	x=x%y
x+=y	x=x+y
x-=y	x=x-y
x<<=y	x=x<<y
x>>=y	x=x>>y
x&=y	x=x&y
x^=y	x=x^y
x|=y	x=x|y

If the *lvalue* of an assignment operator is an array then the *rvalue* must also be an array with the same geometry. The compound assignment operators also support array operands according to the rules for array operands for the parent operation. If

one or both operands are arrays then the operation will be performed element-by-element as described in Section A.6.6.

A.6.18 Comma operator

The comma operator may be used to separate expressions in a single statement. The left operand of the comma operator is evaluated first, then the right operand. The result of the comma expression has the type and value of the last operand evaluated (i.e., the right operand). The comma operator is left associative.

(A.25) *expression* ::= *assignment_expression*(A.23) |
expression , *assignment_expression*

A.7 Constant Expressions

A constant expression is any expression, down to a conditional expression, that may be evaluated to be a constant value at compile time. All constant expressions are evaluated at compile time.

(A.26) *constant_expression* ::= *conditional_expression*(A.22)

A.8 Declarations

Declarations in C_{DSP} are used to declare variables, functions, user defined types, and external references. If the declaration creates storage for the object (a variable *or* a function) then it is also known as a *definition.* Declarations are defined by the following productions.

(A.27) *declaration* ::= *declaration_specifiers* [*init_declarator_list*] ;

(A.28) *declaration_specifiers* ::= *storage_class_specifier*(A.31) [*declaration_specifiers*] |
type_specifier(A.32) [*declaration_specifiers*] |
type_qualifier(A.33) [*declaration_specifiers*]

(A.29) *init_declarator_list* ::= *init_declarator* |

init_declarator_list , *init_declarator*

(A.30) *init_declarator* ::= *declarator*(A.35) |

declarator = *initializer*(A.46)

Declarations may include initializers (see Section A.8.7) that provide the initial value of the storage element if it has local linkage (i.e., an initializer may not be provided for a variable with the `extern` storage-class specifier). Objects with global scope will be initialized only upon program entry if there is an initializer. Objects with local scope but a `static` storage-class specifier will also be initialized only upon program entry if there is an associated initializer. However, if there is an initializer then objects with local scope and an `auto` storage-class specifier will be initialized *every* time the containing scope is entered.

A.8.1 Storage-class specifiers

Storage-class specifiers are used to define the type of storage used for a declared storage object. The `typedef` storage-class specifier is used to define new types and does not allocate any run-time storage. All `typedefed` identifiers have scope limited to the translation unit. All identifiers that are defined by a `typedef` have scope for the remainder of the translation unit.

The `extern` storage-class specifier indicates that the storage for a particular storage object is defined outside of the current translation unit or later within the current translation unit. If the identifier is not defined within the current translation unit then all references to that object must be resolved during the link phase. If the `extern` storage-class specifier is used but the declaration has an initializer then the declaration will be considered a definition. If the identifier is not defined within the

current translation unit then storage will not be allocated in association with the current translation unit.

The `static` storage-class specifier has two meanings. When applied to a storage object with global scope, it indicates that the object will not be made available to other translation units during the link phase. When applied to a storage object with local scope, the object retains its value between successive entries of the containing scope.

The `auto` storage-class specifier indicates that storage for a particular object will be acquired upon entry to the containing scope, and that the storage will be released for reuse upon exiting from the containing scope. Consequently, the storage object may not retain the stored value between successive entries of the containing scope. The `auto` storage-class specifier may not be used for objects with translation unit scope.

(A.31) *storage_class_specifier* ::= `typedef` |
`extern` |
`static` |
`auto`

A.8.2 Type specifiers

The intrinsic type in C_{DSP} are `index`, `char`, `short`, `int`, `long`, `fixed`, and `float`. The `char`, `short`, `int`, `long`, and `index` types are interpreted as integral types. The `index` type has multiple attributes which are discussed in detail in Section A.6.2.

(A.32) *type_specifier* ::= `void` |
`index` |
`char` |
`short` |

`int` |

`long` |

`fixed(`{`char` | `short` | `int` | `long`}`,`*constant*`)` |

`signed` |

`unsigned` |

`float` |

typedef_name(A.45)

The size of the attributes of the `index` type are defined to be whatever is appropriate for the target machine architecture. The size of the remaining integral types are defined the same as in the ANSI C standard: `char` is eight bits (-128 to 127), `short` is sixteen bits (-2^{15} to $2^{15}-1$), `long` is thirty-two bits (-2^{31} to $2^{31}-1$), and the `int` is an implementation defined size between `short` and `long`, inclusive.

The `signed` and `unsigned` attributes may be applied to any of the integral types except `index`. By default, all of the integral types are signed, therefore the `signed` attribute has the effect of a comment. The `unsigned` attribute changes the interpretation of the value from the range $[-2^{N-1}, 2^{N-1}-1]$ to the range $[0, 2^{N}-1]$.

The `fixed` type is a quasi-integral type that is based upon the integral types but is understood to have an implied radix point. The size of the word is derived from one of the existing integral types. The number of fractional bits is user defined. The `fixed` type is a signed type: one bit of the representation is always used as a sign bit. The maximum number of fractional bits for a `fixed` value is one less than the size of the parent integral type. The minimum number of fractional bits is zero, in which case the `fixed` representation would be equivalent to its parent type.

The `float` type is a floating-point number with an implementation dependent representation.

User defined types may be formed using the `typedef` storage class specifier. These types are constructed from the intrinsic types and may be used in the same way as an intrinsic type once defined.

A.8.3 Type qualifiers

There are two type qualifiers: `const` and `volatile`. The `const` qualifier indicates that the identifier being declared cannot be modified. In order for the `const` type qualifier to be meaningful, it is necessary for there to be an initializer in the declaration.

The `volatile` qualifier indicates that the storage object is subject to asynchronous modification by outside sources. Therefore, each access to a storage object declared with the `volatile` qualifier must actually perform the implied access.

(A.33) *type_qualifier* ::= `const` |
`volatile`

(A.34) *type_qualifier_list* ::= *type_qualifier* |
type_qualifier_list type_qualifier

A.8.4 Declarators

(A.35) *declarator* ::= *direct_declarator*

(A.36) *direct_declarator* ::= *identifier* |
(*declarator*) |
direct_declarator[[*constant_expression*(A.26)]] |
direct_declarator ([*parameter_type_list*])

(A.37) *parameter_type_list* ::= *parameter_list* |
parameter_list , ...

(A.38) *parameter_list* ::= *parameter_declaration* |

parameter_list, *parameter_declaration*

(A.39) *parameter_declaration* ::= *declaration_specifiers*(A.28) *declarator* |

declaration_specifiers [*direct_abstract_declarator*(A.44)]

Practical implementation of the function declarator right-hand side elements of the production (A.36) requires that the function declarator be split. To this end, the following productions are used.

(A.40) *direct_declarator* ::= *function_declarator* [*parameter_type_list*])

(A.41) *function_declarator* ::= *direct_declarator* (

The result of substituting productions (A.40) and (A.41) into production (A.36) will be to execute an action associated with the reduction of production (A.41) before any reductions associated with the optional non-terminal *parameter_type_list* can occur.

A.8.5 Type names

(A.42) *type_name* ::= *specifier_qualifer_list* [*direct_abstract_declarator*]

(A.43) *specifier_qualifier_list* ::= *type_specifier*(A.32) [*specifier_qualifier_list*] |

type_qualifier(A.33) [*specifier_qualifier_list*]

(A.44) *direct_abstract_declarator* ::= (*direct_abstract_declarator*) |

[*direct_abstract_declarator*] [[*constant_expression*(A.26)]] |

[*direct_abstract_declarator*] ([*parameter_type_list*(A.37)])

A.8.6 Type definitions

Type definitions may occur in the global scope or within a sub-scope, however, all `typedefed` types have *global* scope. Local identifiers may *not* be declared with the same identifier as that used for a `typedef`'ed type.

(A.45) *typedef_name* ::= *identifier*

A.8.7 Initialization

Initializers provide for the initialization of variable and constant data storage objects within the declaration. All initializers take the form of a declared *lvalue*, an assignment operator, and the value to use for initialization on the right. The value (or values) used in an initializer must be a constant expression.

(A.46) *initializer* ::= *assignment_expression*(A.23) |
{*initializer_list*} |
{*initializer_list*, }

(A.47) *initializer_list* ::= *initializer* |
initializer_list, *initializer*

Array initializers may be created using comma-separated lists enclosed in braces. Array initializers should be the same size as or smaller than the array to be initialized; initializers that are larger than the array to be initialized are not allowed. If an array initializer is present but it is smaller than the array being initialized, the remainder of the array will be set to zero.

A.9 Statements

Statements are the elements of the translation unit that are used to generate object code. Statements are executed in sequence except where flow is explicitly altered by branching. All statements are found in the bodies of functions. While the C_{DSP} language, like its namesake, supports a restricted `goto` statement, its use is discouraged since it tends to make code both less manageable for the human programmer and the compiler. A structured programming style is encouraged by the

rich control flow statement options and the structuring of all C_{DSP} programs as lists of functions.

(A.48) *statement* ::= *labeled_statement*(A.49) |
compound_statement(A.50) |
expression_statement(A.53) |
selection_statement(A.54) |
iteration_statement(A.55) |
jump_statement(A.56)

A.9.1 Labeled statements

Labels are identifiers that are prepended to statements using a colon to delimit the identifier from the labeled statement. These labels are used as targets for the `goto` statement (see Section A.9.6). The scope of a labeled statement is local; the label is not visible from outside of the containing function.

(A.49) *labeled_statement* ::= *identifier* : *statement*(A.48)

A.9.2 Compound statements

Compound statements are groupings of statements that may be used anywhere a single statement may be used. Compound statements may contain declarations and executable statements per Production (A.50), although neither is required.

(A.50) *compound_statement* ::= {[*declaration_list*][*statement_list*]}

(A.51) *declaration_list* ::= *declaration*(A.27) |
declaration_list declaration

(A.52) *statement_list* ::= *statement*(A.48) |
statement_list statement

A.9.3 Expression statements

Expression statements are statements that only contain expressions. All expression statements are terminated by a semi-colon. An expression statement may be empty, denoted by the required terminating semi-colon. Such an empty statement is referred to as a *null statement.* A null expression may be used anywhere a statement is required.

(A.53) *expression_statement* ::= [*expression*(A.25)] ;

A.9.4 Selection statements

The `if` and `if-else` statements are used to evaluate expressions and execute code depending upon the result of the evaluated expression. The `if` statement (Production (A.54)) first evaluates the expression contained in parentheses. The expression must be a scalar. If the expression evaluates to be true (non-zero) then the target statement is executed, otherwise, if the expression evaluates to be false (zero) then the target statement is not executed. In the case of the `if-else` statement, the first statement is executed if the expression evaluates to be true (non-zero) otherwise the second statement executes. The control flow for the `if` and `if-else` selection statements is shown in Figure A.2.

(A.54) *selection_statement* ::= `if (`*expression*(A.25)`)` *statement*(A.48) |
`if (`*expression*`)` *statement* `else` *statement*

The target statements for the `if` and `if-else` selection statements may be any statement, including another `if` or `if-else` statement. This capability introduces an ambiguity whose resolution is not apparent from Production (A.54), namely the problem of the dangling `else`. In particular, in the structure `if-if-else` it is not clear whether the `else` associates with the first or second `if`. By definition, the `else` will always associate with the closest `if`.

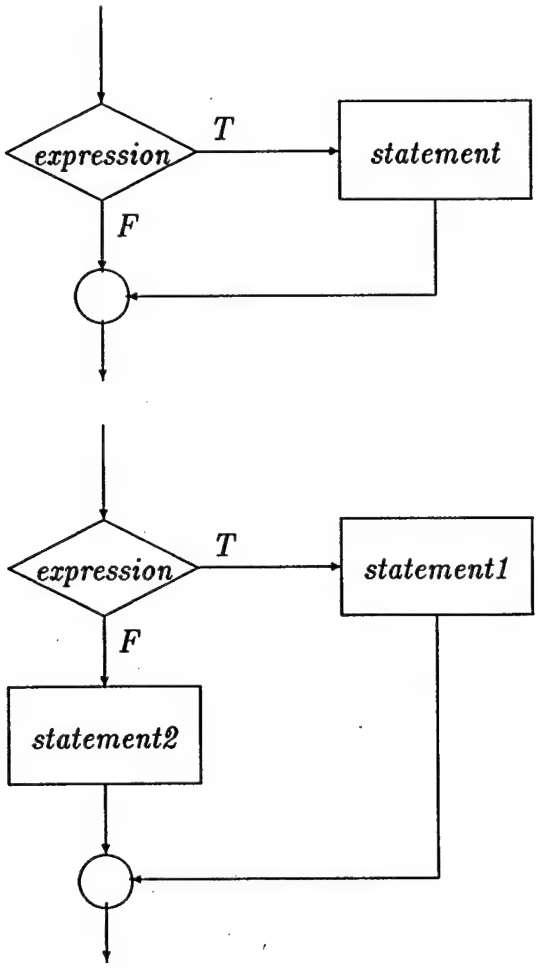


Figure A.2: Control Flow For the `if` and `if-else` Statements

A.9.5 Iteration statements

The iteration statements are the C_{DSP} statements that are used for looping constructs.

(A.55) *iteration_statement* ::= `while` (*expression*(A.25)) *statement*(A.48) |
`do` *statement* `while` (*expression*) `;` |
`for` ([*expression*]; [*expression*]; [*expression*]) *statement* |
`dopar` ([*expression*]; [*expression*]; [*expression*]) *statement*

The `while` statement is the only looping statement that is needed to implement all sequential looping constructs. The expression in the parentheses is evaluated and

if it is true (non-zero) then the target statement is executed. The flow of execution of the `while` statement is illustrated in Figure A.3.

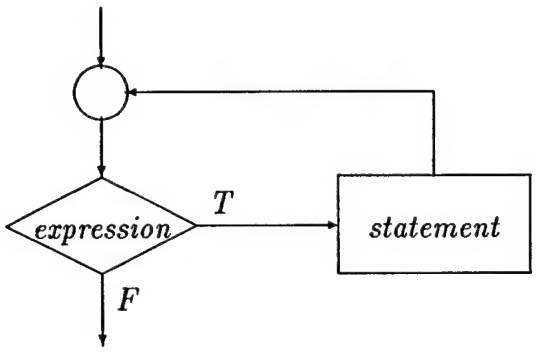


Figure A.3: Control Flow For the `while` Statement

The `do-while` statement is similar to the `while` statement except that it executes its statement before evaluating the conditional expression that controls looping. A flow diagram of the `do-while` statement is shown in Figure A.4.

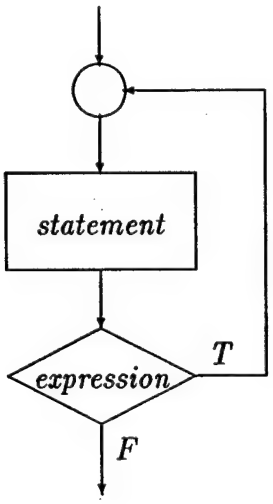


Figure A.4: Control Flow For the `do-while` Statement

The `for` statement has the classic elements of the for-loop structure. There is an expression that is evaluated upon entry that serves to initialize any needed iteration control variables, a condition expression that is tested once per iteration, a statement that is executed once per iteration that serves as the "payload" of the loop, and a final

expression that is evaluated after the statement is executed to update the iteration variables. The `for` statement in the C_{DSP} language works like its C equivalent. A flow diagram of the execution of a `for` statement is given in Figure A.5.

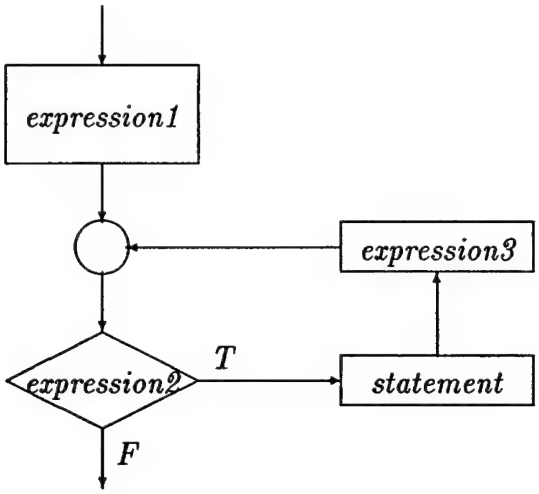


Figure A.5: Control Flow For the `for` Statement

The `dopar` statement is a parallel loop statement. The `dopar` statement is defined to execute as if each iteration has a separate copy of all data objects that will be accessed within the loop. Therefore, there is no flow dependence between iterations of the `dopar` loop. If there is a data access conflict between iterations then the resulting behavior is undefined. As such, it is best to avoid data access conflicts between iterations. There should be no data dependencies from the body of the loop to the iteration variable(s). The `dopar` loop executes as if the iteration variables are all computed first, and each loop "iteration" (statement) begins execution with a separate copy of the iteration variable(s), that is, each iteration is forked. The `dopar` executes as if when all iterations have completed a join is performed. A block diagram illustrating the flow of a `dopar` statement is given in Figure A.6.

Since C_{DSP} functions are not reentrant, any function calls within a `dopar` loop will result in sequential execution of the loop statement. In a future revision of the language, it would be worthwhile to provide either function reentrancy or function

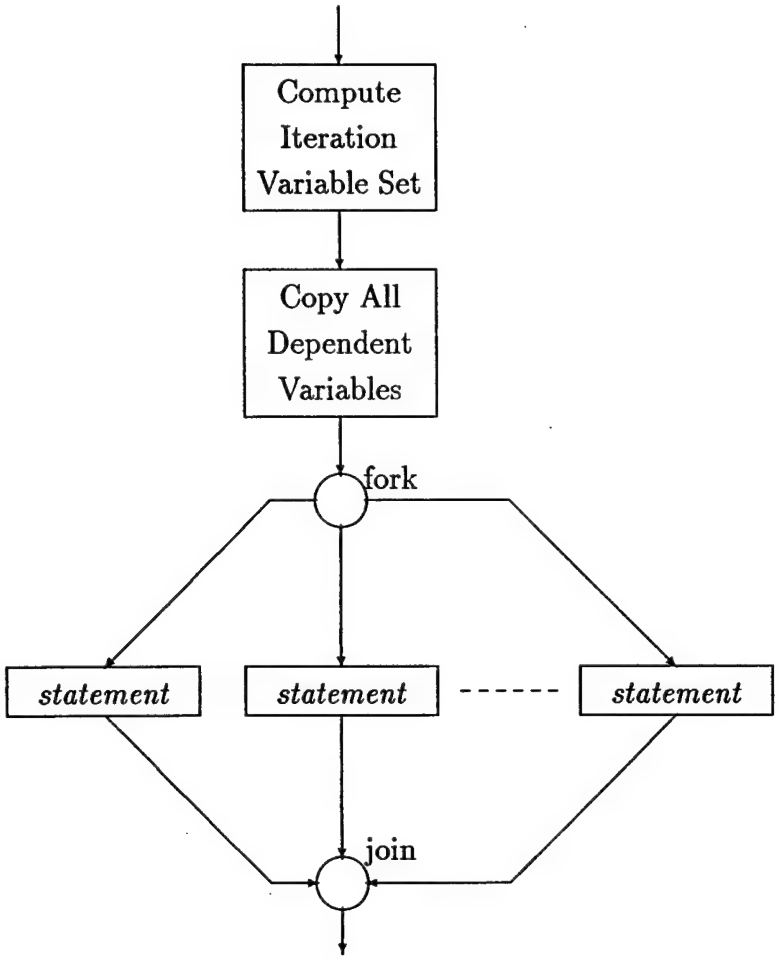


Figure A.6: Control Flow For the dopar Statement

locking (so that only one thread of execution may be in the function at any time) to simulate function reentrancy.

A.9.6 Jump statements

The jump statements are used to alter program flow outside of the normal application of the structured iteration statements. The goto jump statement is used to branch to another statement within the the local scope of a function. It is provided primarily as a porting aid; many legacy applications are highly dependent upon an unrestricted goto. The availability of the goto allows manual translation and even machine translation of existing code.

The `continue` and `break` statements are used to alter the flow of control within an iteration statement. The `break` statement simply causes the loop to exit directly when the `break` statement is encountered. In the case of nested loops, the `break` statement does not cause all of the loops to be broken but rather the one directly containing the `break` statement.

The `continue` statement causes the currently executing iteration of a loop to exit and causes the program flow to proceed to the next iteration. Outside of loops the `break` and `continue` statements have no effect upon program control flow.

The `return` statement causes the current function to exit and the control of the program to return to the calling program. The `return` statement may be given with or without an optional expression. If the expression is not empty then the function will return the value computed in the expression. The type of this returned value must be consistent with the declared return type of the containing function.

(A.56) *jump_statement* ::= `goto` *identifier* `;` |
`continue ;` |
`break ;` |
`return` [*expression*(A.25)] `;`

APPENDIX B
M-FILES

B.1 DFT Code

B.1.1 rpdft.m

```
% Rader Prime DFT Function.
% Author: Jon Mellott
% Date:   10-18-93
% Description:
%   This function performs the Rader Prime DFT on a complex data
% sequence. Circular convolution is performed by multiplying the
% fft's of the sequences of interest.
%
% Modified 5/25/94W. Indexing bug fixed.
% Arguments:
% cx -- Complex input data.
% dftl -- Length of dft; must be prime.
% pelmt -- Primitive element of GF(dftl).
function V = rpdft(cx,dftl,pelmt)
% Prepare to do Rader prime DFT
% Create permutation matrix to scramble input data.
Pin=zeros(dftl-1);
Pout=Pin;
for I=0:dftl-2
Pin(I+1,rem(pelmt^(rem(dftl-I-1,dftl-1)),dftl))=1;
Pout(I+1,rem(pelmt^I,dftl))=1;
end;
% Generate circular convolution sequence.
F=zeros(dftl-1,1);
for I=0:dftl-2
F(I+1)=exp(-i*2*pi*rem(pelmt^I,dftl)/dftl)-1;
end
% Prepare to do circular convolution using products of DFTs.
F=fft(F);
% Do Rader prime DFT.
V=ifft(fft(Pin*cx(2:dftl)).*F);
% Unscramble the output.
V=Pout.'*V;
% Add the DC term.
V=sum(cx(1:dftl))+[0;V];
end
```

B.1.2 gtdft.m

```
% Good-Thomas DFT Function.
% Author: Jon Mellott
% Date:   6-2-94
% Description:
%
% Arguments:
% x -- data
% cf -- CRT configuration matrix.
```

```
function X=gtdft(x,cf)
% Extract the number of primes.
[L t]=size(cf); % We only care about L.
% Extract the vector length from CRT configuration matrix.
M=prod(cf(:,1));
% For each p_{i}, take all DFT's of length p_{i} for GT-DFT.
for i=1:L
  % Compute prime list/crt configuration permutation matrix.
  P=zeros(L);
  for j=1:i-1
    P(j,j)=1;
  end
  for j=i+1:L
    P(j-1,j)=1;
  end
  P(L,i)=1;
  % Permute the CRT configuration matrix.
  cfp=P*cf;
  % Establish an index vector of length L-1.
  I=zeros(L-1,1);
  % Set the done flag to zero.
  bDone=0;
  % Perform DFT's
  while (bDone==0)
    % Create an index set for the data vector.
    J=zeros(cfp(L,1),1);
    for j=0:cfp(L,1)-1
      J(j+1)=crt([I;j],cfp)+1;
    end
    % Perform a DFT along the elements of x indexed by J.
    x(J)=fft(x(J));
    % Increment the index vector.
    j=1;
    bDone2=0;
      while (bDone2==0)
        I(j)=I(j)+1;
        if (I(j)==cfp(j,1))
          I(j)=0;
          if (j>=L-1)
            bDone=1;
            bDone2=1;
          end
        else
          bDone2=1;
        end % End If-Else
        j=j+1;
      end % End While
    % Done incrementing the index vector!
  end
end
% Permute transformed vector to correct order.
X=zeros(M,1);
for i=1:M
  X(rem(rem((i-1)*ones(L,1),cf(:,1)).'*cf(:,2),M)+1)=x(i);
end
end
```

B.2 CRT Code

B.2.1 crtconf.m

```
% CRT Configuration Function.
% Author: Jon Mellott
% Date:   10-18-93
% Description:
%   This function computes the m_{i} and m_{i}^{-1} factors need
```

```
% for the CRT. The results are arranged in a matrix where the
% first column is the prime list, the second column is the
% list of m_{i}'s, and the third column is the list of m_{i}^{-1}'s.
%
% Modified 5/31/94T:
% A fourth output column has been added, a list of the generators
% of each GF(p_{i})\{0}. This is useful for GE/LRNS and the Rader prime
% DFT, especially when used in the Good-Thomas DFT.
% Arguments:
% plist -- Prime list vector.
function C=crtconf(plist)
% Compute the product of the prime list.
M=prod(plist);
% Compute mi=M/pi list.
m=M*ones(max(size(plist)),1)./plist;
% Compute the inverses of each mi in Zpi.
mi=zeros(max(size(plist)),1);
for I=1:max(size(plist))
  J=1;
  while (rem(J*m(I),plist(I))~=1)
    J=J+1;
  end
  mi(I)=J;
end
% Compute the generators for each pi.
gi=zeros(max(size(plist)),1);
for I=1:max(size(plist))
  gi(I)=gen(plist(I));
end
% Build CRT configuration matrix.
C=[plist m mi gi];
end
```

B.2.2 gen.m

```
% Generator Function.
% Author: Jon Mellott
% Date:   5-31-94
% Description:
%   This function finds a generator for GF(p)\{0}.
%
% Arguments:
% p -- Prime number.
function alpha=gen(p)
% Initialize done flag to zero.
bDone=0;
% Initialize generator.
a=1;
% Search until done.
while (bDone==0)
  % Increment generator.
  a=a+1;
  % Initialize ones count.
  iCount=0;
  % Initialize exponent.
  x=a;
  % Search for a generator.
  for i=2:p-1
    x=rem(x*a,p);
    if (x==1) % Increment ones count.
      iCount=iCount+1;
    end
  end
  % Check for found generator.
  if (iCount==1)
```

```
    bDone=1;
    alpha=a;
  else
    % Check for non-existence of generator.
    if (a==p-1)
      bDone=1;
      alpha=0;
    end
  end
end
end
```

B.2.3 crt.m

```
% CRT Function.
% Author: Jon Mellott
% Date:   10-18-93
% Description:
%   This function converts a residue n-tuple to an integer using the
% CRT.
%
% Arguments:
% ntuple -- The residue vector to be converted.
% confmat -- The configuration matrix produced by the function confmat.
function X=crt(ntuple,confmat)
X=rem(sum(confmat(:,2).*rem(confmat(:,3).*ntuple,confmat(:,1))),prod(confmat(:,1)));
end
```

APPENDIX C
TYPOGRAPHICAL NOTES

> Preparation of the thesis is one aspect of the training in the mature and responsible scholarship expected of the candidate. Time devoted to careful attention to form, style, and mechanics should not be regarded as time wasted in mechanical compliance with administrative regulations. The thesis is a public and permanent record of the candidate's professional attainment and reveals the quality and standards of his or her workmanship.
>
> From the *University of Florida Record*, 1960-1961

This dissertation was produced using LaTeXversion 2.09. The primary typeface used in this dissertation is Computer Modern. Certain mathematical fonts used in this dissertation were obtained from the American Mathematical Society. Figures in this dissertation were produced using a combination of CorelDRAW!, Corel Photo-Paint, Adobe Illustrator, and xfig. The photographs were scanned with a Hewlett-Packard ScanJet 4C/T, at a resolution of 600 dots per inch (dpi) by 256 gray levels. The photographs were manipulated and converted to 600 dpi black and white images using Corel Photo-Paint. This document was printed using dvips to generate Adobe PostScript that was sent to a Hewlett-Packard LaserJet 4SiMX with a resolution of 600 dpi.

The University of Florida's dissertation formatting requirements are clearly designed for a typewritten dissertation. In recent years some slight modifications to the formatting requirements have been made, finally allowing the use of bold text for headings and a monospaced font for computer listings. Despite these small steps, the double spaced format is inefficient, wasting paper and shelf space, and aesthetically

unattractive. To the author it seems ironic that the quotation given above is displayed in the University of Florida's *Guide for Preparing Theses and Dissertations.*

REFERENCES

[1] G. D. Hutcheson and J. D. Hutcheson, "Technology and economics in the semiconductor industry," *Scientific American*, vol. 274, pp. 54–62, Jan. 1996.

[2] H. S. Stone, *High Performance Computer Architecture.* Reading, Massachusetts: Addison-Wesley, 2nd ed., 1990.

[3] S. E. Schuster, "Multiple word/bit line redundancy for semiconductor memories," *IEEE Journal of Solid-State Circuits*, vol. SC-13, pp. 698–703, Oct. 1978.

[4] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing.* Englewood Cliffs, New Jersey: Prentice-Hall, 1975.

[5] G. H. Golub and C. F. van Loan, *Matrix Computations.* Baltimore: Johns Hopkins University Press, 2nd ed., 1989.

[6] S. L. Marple, *Digital Spectral Analysis with Applications.* Englewood Cliffs, New Jersey: Prentice-Hall, 1987.

[7] R. E. Blahut, *Fast Algorithms for Digital Signal Processing.* Reading, Mass.: Addison-Wesley Publishing Company, 1985.

[8] J. Cooley and J. Tukey, "An algorithm for machine calculation of complex Fourier series," *Mathematical Computation*, vol. 19, pp. 297–301, 1965.

[9] I. Good, "The relationship between two fast Fourier transforms," *IEEE Trans. Computers*, vol. C-20, pp. 310–317, Mar. 1971.

[10] C. M. Rader, "Discrete Fourier transforms when the number of data samples is prime," *Proceedings of IEEE*, pp. 1107–1108, 1968.

[11] F. J. Taylor, "An RNS discrete Fourier transform implementation," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 38, pp. 1386–1394, Aug. 1990.

[12] G. S. Zelniker and F. J. Taylor, "Prime blocklength discrete Fourier transforms using the polynomial residue number system," in *Proc. Twenty-Fourth Asilomar Conf. on Signals, Systems, and Computers*, 1990.

[13] J. D. Mellott, M. Lewis, and F. J. Taylor, "ASAP – a 2D DFT VLSI processor and architecture," in *Proc. IEEE International Conf. on Acoustics, Speech, and Signal Processing*, (Atlanta), 1996.

[14] J. D. Mellott, M. Lewis, and F. J. Taylor, "ASAP – a 2D DFT VLSI processor and architecture," in *Proc. IEEE International Symposium on Circuits and Systems*, (Atlanta), 1996.

[15] Texas Instruments, *TMS320C50 User's Manual.* Dallas: Texas Instruments, 1993.

[16] Motorola, *DSP56000/DSP56001 User's Manual.* Phoenix: Motorola, 1990.

[17] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitative Approach.* San Francisco: Morgan Kaufmann Publishers, 2nd ed., 1996.

[18] A. Smith and J. Lee, "Branch prediction strategies and branch-target buffer design," *Computer*, vol. 17, pp. 6–22, Jan. 1984.

[19] H. M. Levy and R. H. Eckhouse, *Computer Programming and Architecture: the VAX.* Digital Press, 2nd ed., 1989.

[20] J. Fisher and B. Rau, "Instruction-level parallel processing," *Science*, vol. 253, pp. 1233–1241, Sept. 1991.

[21] M. Gokhale and W. Carlson, "Introduction to compilation issues for parallel machines," *Journal of Supercomputing*, vol. 6, pp. 283–314, Dec. 1992.

[22] G. Blanck and S. Krueger, "The SuperSPARC microprocessor," in *Proc. IEEE Computer Society International Conference*, pp. 136–141, 1992.

[23] D. F. Snelling and G. K. Egan, "Comparative study of data-flow architectures," in *IFIP Transactions A: Computer Science and Technology*, vol. A-50, 1994.

[24] R. P. Colwell, R. P. Nix, J. J. O'Donnel, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," *IEEE Trans. Computers*, vol. 37, pp. 967–979, Aug. 1988.

[25] M. A. Schuette and J. P. Shen, "Instruction-level experimental evaluation of the Multiflow TRACE 14/300 VLIW computer," *Journal of Supercomputing*, vol. 7, pp. 249–271, May 1993.

[26] J. Gray, A. Naylor, A. Abnous, and N. Bagherzadeh, "VIPER: A VLIW integer microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 28, pp. 1377–1382, Dec. 1993.

[27] A. Abnous and N. Bagherzadeh, "Pipelining and bypassing in a VLIW processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 658–664, June 1994.

[28] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Computers*, vol. 30, pp. 478–490, July 1981.

[29] J. Mick and J. Brick, *Bit-Slice Microprocessor Design.* New York: McGraw-Hill, 1980.

[30] C. Babcock, "Silicon marriage: HP/Intel venture," *Computerworld*, vol. 28, p. 6, July 1994.

[31] W.-M. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Change, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for VLIW and superscalar compilation," *Journal of Supercomputing*, vol. 7, pp. 229–248, May 1993.

[32] E. Arnould, H. T. Kung, O. Menzilcioglu, and K. Sarocky, "A systolic array computer," in *Proc. IEEE International Conf. on Acoustics, Speech, and Signal Processing*, 1985.

[33] H. T. Kung *et al.*, "iWarp: An integrated solution to high-speed parallel computing," *IEEE Trans. Computers*, vol. 38, pp. 330–339, Sept. 1988.

[34] R. Simar, "The TMS320C40: A DSP for parallel processing," in *Proc. IEEE International Conf. on Acoustics, Speech, and Signal Processing*, vol. 2, pp. 1089–1092, 1991.

[35] R. Weiss, "TI multiprocessor chip peaks at 2 billion operations/sec," *EDN*, vol. 39, pp. 67–68, Mar. 1994.

[36] S. Haykin, *Adaptive Filter Theory.* Englewood Cliffs, New Jersey: Prentice-Hall, 2nd ed., 1991.

[37] J. D. Mellott, "The Gauss machine: A GEQRNS DSP systolic array," Master's thesis, University of Florida, 1993.

[38] J. D. Mellott, J. C. Smith, and F. J. Taylor, "The Gauss machine — a Galois-enhanced quadratic residue number system sysotlic array," in *Proc. IEEE 11th Symposium on Computer Arithmetic*, (Windsor, Ontario), pp. 156–162, 1993.

[39] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective.* Reading, Mass.: Addison-Wesley Publishing Company, 1985.

[40] W. K. Jenkins, "The design of error checker for self-checking residue number arithmetic," *IEEE Trans. Computers*, vol. 32, pp. 388–396, Apr. 1983.

[41] J. V. Krogmeier and W. K. Jenkins, "Error detection and correction in quadratic residue number systems," in *26th Midwest Symposium on Circuits and Systems*, 1983.

[42] S. Y. Kung, "VLSI array processors," *IEEE ASSP Magazine*, pp. 4–22, July 1985.

[43] M. Griffin, F. J. Taylor, and M. Sousa, "New scaling algorithms for the chinese remainder theorem," in *Proc. 22nd Asilomar Conf. on Signals, Syst., and Computers*, 1988.

[44] M. Griffin, M. Sousa, and F. J. Taylor, "Efficient scaling in the residue number system," in *Proc. IEEE International Conf. on Acoustics, Speech, and Signal Processing*, 1989.

[45] G. Zelniker and F. J. Taylor, "A reduced complexity finite field ALU," *IEEE Trans. on Circuits and Systems*, vol. 38, pp. 1571–1573, Dec. 1991.

[46] F. J. Taylor, J. Mellott, J. Smith, and G. Zelniker, "The Gauss machine — a DSP processor with high RNS content," in *Proc. IEEE International Conf. on Acoustics, Speech, and Signal Processing*, (Toronto), 1991.

[47] M. Wolfe, *High Performance Compilers for Parallel Computing.* Reading, Massachusetts: Addison-Wesley, 1996.

[48] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools.* Reading, Massachusetts: Addison-Wesley, 1986.

[49] A. N. S. Institute, ed., *Programming Languages: C.* New York: American National Standards Institute, 1990.

[50] D. M. Ritchie, "Unix time-sharing system: A retrospective," *The Bell System Technical Journal*, vol. 57, pp. 1947–1969, July/August 1978.

[51] D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "The C programming language," *The Bell System Technical Journal*, vol. 57, pp. 1991–2019, July/August 1978.

INDEX

BIOGRAPHICAL SKETCH

Jonathon D. Mellott was born on November 20th, 1968. He graduated from the University of Florida in 1990 with a Bachelor of Science in Electrical Engineering with a minor in mathematics. He obtained a Master of Science in electrical engineering from the University of Florida in 1993. He has been a consultant to industry, a computer and information science instructor at Santa Fe Community College, has held a Martin Marietta Corporate Scholarship, and is a member of Eta Kappa Nu, the Institute of Electrical and Electronic Engineers, and the Association for Computing Machinery.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Fred J. Taylor, Chairman
Professor of Electrical and
Computer Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Donald G. Childers
Professor of Electrical and
Computer Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Jose C. Principe
Professor of Electrical and
Computer Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Jian Li
Associate Professor of Electrical
and Computer Engineering

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Rick L. Smith
Associate Professor of Mathematics

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

December, 1997

Winfred M. Phillips
Dean, College of Engineering

Karen A. Holbrook
Dean, Graduate School

Appendix B: Wireless Local Area Network and Channel Modeling at 2.4 GHz

WIRELESS LOCAL AREA NETWORK & CHANNEL MODELING AT 2.4 GHZ

WRITTEN BY GAEL CHAMPION, WILLIAM AGASSOUNON, & STUART LOPATA

ACKNOWLEDGEMENT

We would like to thank Dr Fred J. Taylor Director of the High Speed Digital Architecture Laboratory of the University of Florida for all the means he gave us to achieve this work.

Dr David Chester, Dr William Hortos, Dr Clay Frace and all the staff of the wireless LAN research department of Harris Semiconductor at Melbourne (Florida) for all the equipment and the hardware they provided us with and their advice.

Dr Uwe Meyer-Baese, Dr Iztok Koren, of the HSDA lab.

Mr. Hicham Bouzekri of the L IST lab.

TABLE OF CONTENTS

TABLE OF ILLUSTRATIONS

INTRODUCTION

Nowadays, wireless communications systems are growing up faster and faster in order to match the work and people mobility. Internet also gave people the opportunity of discovering at home what is located elsewhere, everywhere in the world.

The current technology, which looked like science fiction 15 year ago, needs more mobility if they can wireless phone, they are, most of the time, still obliged to connect their computer to a network in order to get an outdoor access. As you keep your cellular phone always with you, what about moving with your laptop without loosing any connection to your LAN (Local Area Network)?

Here is the challenge that many telecommunications companies are trying to achieve. For the moment, almost all the wireless applications were low speed transmission, just in order to transport voice sampled under 10 KBPS. The problem was that to get a comfortable network connection, people really need more speed and a larger bandwidth.

That is why many WLAN products are now proposed by a few companies with high bit rate up to several MBPS. Most of them use the 2.4 GHz of the ISM band (Industrial/Science/Medical band) like Harris Corporation which developed a complete PCMCIA Card WLAN solution (this is the object of this study).

Indeed, this band - whose one main advantage is to be unlicensed - is very sensitive to fading and multipath. Moreover, this was for the moment almost not used (except by microwave ovens) and that is why it is still there are not a lot of modeling yet

Hence, a complete simulation of such a transmission including a channel modeling (for the main conditions of propagation) can be very useful in order to get an idea of the best locations of the transceivers and of the number of repeaters eventually required.

Lastly, these new communications systems are based on Spread Spectrum technologies in order to reduce interference with other systems at the same frequency. That is why, a part of this report is also dedicated to this increasingly used technology.

PROPAGATION PATH LOSS MODELING

1. Introduction

Much work has been done to statistically characterize multipath propagation inside building in the 800 MHz to 5.8 GHz frequency range (See references [1]-[11]). Buildings vary greatly in size, shape, and type of construction materials. The statistics of propagation measurements vary greatly from building to building and only conclusion related to a particular building type can be made.

The purpose of this work is to model propagation losses within buildings at UHF frequencies. Specifically, it presents results of measurements taken at 2.4 GHz mainly in the new engineering building of the University of Florida, and it elaborates an empirical attenuation model based on these measurements.

A statistical model of the simplest form relates the average path loss to the log of distance. The distance between the transmitter and receiver measured in three dimensions, and n the mean path loss exponent, indicates how fast path loss increases with distance (n=2 for free space).

Work in [2] shows that in multifloored buildings, more accurate prediction is possible when the parameter n is viewed as a function of the number of floors between transmitter and receiver.

The path loss model developed in [1] to predict attenuation in multifloored building is used. For measurement when the transmitter and receiver are located on the same floor, we developed an alternative path loss model ((3)) to quantify the additional path loss caused by walls (concrete and soft obstructions) between the transmitter and receiver.

This paper is organized as followed. Section II deals with the measuring equipment and the experimental site. Section III presents the measurements data and associates the measured path losses to simple algebraic relationship of the type $y=a+b\log_{10}x$ and gives an overall view of the mean RMS delay spread of each location measurements. The maximum delay time spread is the total time interval during which reflections with significant energy arrive. The RMS delay spread is the standard deviation value of the delay of reflections, weighted proportional to the energy of the reflected waves. It is given by the mean of the square root of the second central moment of the power delay spread profile and is calculated by the matlab code ***delay.m*** described in annex. A threshold is set at 40 dB below the peak path loss in the time domain to define the noise floor. Section IV presents a summary of the path loss and RMS delay spread of different measurements. Section V is devoted to discussion and conclusions and to an evaluation of how well the modeling fit the data.

2. Experimental conditions

The experiments have been conducted with a continuous wave emission at a central frequency of 2.4 GHz in a bandwidth of 110 MHz. A transmit power of 10 dBm for most measurements was used and 15 dBm for the experiments between two floors. The source was incorporated in the network analyzer that was also used as receiver, and this permitted measurements down to a threshold of -110 dBm.

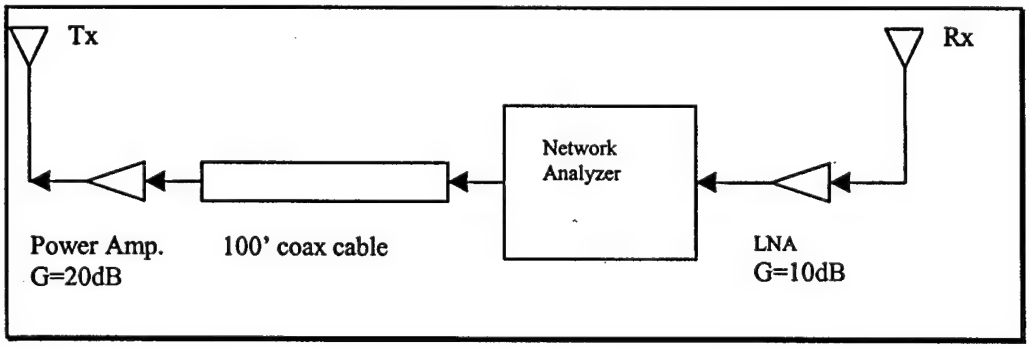
The wave is transmitted by an omnidirectional half-wave dipole antenna (MACOM) with 1.9 dB gain.

The analyzer can instantaneously measure signal strength between 0 and -110 dBm over a 110 MHz bandwidth. This is on the order of the maximum dynamic range expected for emerging Personal Communication Network (PCN) which will be deployed within buildings the next several years.

Most of the measurements have been taken in the Electrical Engineering department at the fourth floor of the new engineering building of the University of Florida (see blueprint). That building offers a mixture of offices, laboratories and classrooms and dates back to 1997. The floors are in reinforced concrete. Internal walls are made of lightweight concrete and materials about 13 cm thick. The external walls are made of heavy concrete and bricks. Each floor is 4.5 m high with drop soft ceiling at 3.8 m.

The equipment:

The transmitting and receiving antennas are set as described in fig.1. The cable is a coax of 100 feet long, the LNA and the power amplifier are both of Mini-Circuits of 10 and 20 dB gain respectively.



(1) Channel modeling experimental conditions

3. Path loss prediction models

A model used in works in [1]-[6], [8], [9], [12] indicates that mean path loss increases exponentially with distance. That is, the mean path loss is a function of distance to the ***n*** power.

$$\overline{PL}(d)_{[dB]} \alpha \left(\frac{d}{do}\right)^{n}$$

Equation (1)

where:

- ***PL*** is the mean path loss
- ***n*** is the mean path loss exponent which indicates how fast path loss increases with distance
- ***do*** is a reference distance
- ***d*** is the transmitter-receiver separation distance.

When plotted on a log-log scale, the power-distance relationship resembles a straight line.

Absolute mean path loss, in decibels, is defined as the path loss from the transmitter to the reference distance ***do***, plus the additional path loss described by [1] in decibels.

$$\overline{PL}(d)_{[dB]} = \overline{PL}(do)_{[dB]} + 10 \times n \times \log_{10}\left(\frac{d}{do}\right)$$

Equation (2)

For our measurements, a one meter reference distance was chosen and we measured ***PL(do)*** = *40dB*.

In [1], [2] path loss is shown to have a log-normal distribution. Assuming this distribution for our data, we determined the mean path loss exponent ***n*** and standard deviation σ (in decibels), which are viewed as parameters that are a function of building type and number of floors between transmitter and receiver.

The standard deviation provides a quantitative measure of the accuracy of the model used to predict the path loss for a given environment. The path loss at a T-R separation of ***d*** meters is then given by:

$$PL(d)_{[dB]} = \overline{PL}(d)_{[dB]} + X_{\sigma[dB]}$$

Equation (3)

where X_σ is a zero mean log-normally distributed random variable with standard deviation σ in decibels.

Linear regression was used to compute values of the parameters ***n*** and σ in a Minimum Mean Square Error (***MMSE***) sense for the measured data.

The data are grouped by propagation types to provide some insight on how the path loss changes based according to the environment. Effective groupings will usually bring about smaller deviations given by:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}\left(PLi - PLi\ (di)\right)^2$$

Equation (4)

where :

- ***PLi*** is the measured path loss
- ***PLi (di)*** is the predicted path loss
- ***N*** is the number of values computed.

A. Propagation with Line Of Sight LOS)

Table I

Environment & Tx locations (see blueprint)	***n***	***σ[dB]***	***Mean RMS Delay spread (ns)***	***Number of measurements***
All locations	1.6	3.75	60	632
Tx in A1 (499B)	1.55	3.08	87	64
Tx in B1 (499B)	1.85	2.31	92	64
Tx in C (499B & 499G)	1.65	3.57	31	64
Tx in A2 (499B)	1.4	2.52	59	64
Tx in A3 (499B)	1.55	2.93	55	64
Tx in B2 (499B)	1.55	2.73	74	63
Tx in B3 (499B)	1.65	2.58	59	63
Tx in A4 (499B)	1.45	4.21	41	93
Tx in D (in 599G)	2	3.12	46	93

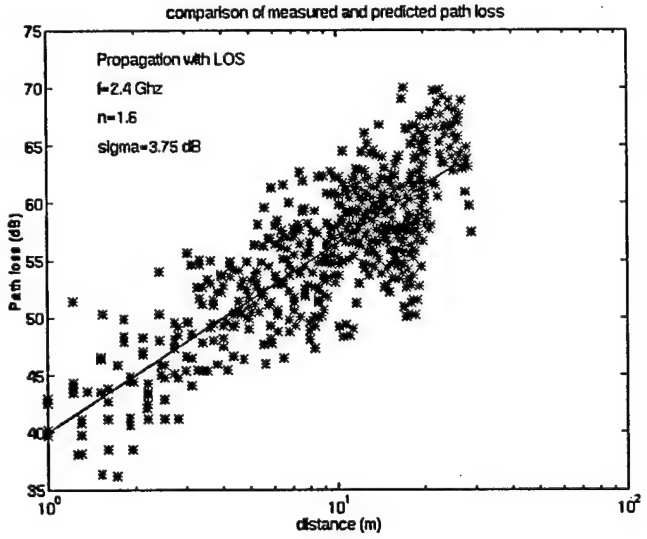
Table I summarizes the mean path loss exponents, standard deviations about the mean for different LOS environments. The table also includes the mean delay spread and the number of measurements.

From Table I, it can be seen that the parameters for path loss prediction with LOS in the building are n=1.6 and σ=3.75. This value of σ is typical for data collected from different Tx positions, it is influenced by the size of the walls and hallways that decide the Fresnel zone. The standard deviation indicates that about ***82%*** of actual measurements are within ***±5 dB*** of the predicted mean path loss.

Scatter plot of path loss versus T-R separation distance for propagation with LOS is given in fig.2. The dashed line indicates the best mean path loss model in a MMSE sense for the data presented in the scatter plot. This figure shows that the mean path loss increases with distance at the 1.6 power with a standard deviation of 3.75 dB for the propagation with LOS in the building, while model is accurate to within ±5 dB for more than 82% of all measurements.

This path loss exponent (n=1.6) is less than 2 (the free space path loss exponent) because of the extra gain caused by multiple reflections indoor. In the following, we assume this exponent (n=1.6) is the indoor propagation path loss exponent as log as no obstacles lie between the transmitter and receiver.

Metallic doors at the end of hallways 499B,499G, 599B and 599G have an important effect on the RMS delay spread as shown by Table I. When the transmitter and receiver are located away from these doors, the delay spread is unaffected by them.



(2) Propagation with LOS path loss prediction

B. Propagation through floors: Floor Attenuation Factor (FAF)

Floors in the new engineering building of the University of Florida are made from reinforced concrete. Each floor is 4.4 m high with a suspended soft ceiling at 2.4 m in the hallway and 3 m in the rooms.

In this section, the path loss in multifloored environments is predicted using the model

developed in [2]. This model uses a mean path loss exponent that is the same for the building and a constant floor attenuation factor (FAF) that is a function of number floors and the building type. This factor gets added to the mean path loss predicted by the previous model (2):

$$\overline{PL}(d)_{[dB]} = \overline{PL}(do)_{[dB]} + 10 \times n(same\_floor) \times \log_{10}\left(\frac{d}{do}\right) + FAF_{[dB]}$$

Equation of model (2)

where:

- ***d*** is in meters
- ***PL(do)*** =40 dB at 2.4 GHz
- ***n*** =1.6 (path loss exponent for the building)

Table II (with n=1.6)

Locations	***FAF[dB]***	***σ[dB]***	***Mean RMS delay spread (ns)***	***Number of measurements***
All locations	23	5.06	37	358
Tx in 3A & Rx in 499B	28	1.34	38	32
Tx in 3B & Rx in 499B	26	1.88	42	32
Tx in 3A & Rx in 499C	27	1.92	55	32
Tx in 5A & Rx in 499B (fire blocker open)	23	6.55	32	64
Tx in 5A & Rx in 499B (fire blocker shut)	22	5.77	36	64
Tx in 5B & Rx in 499B	19	2.71	29	37
Tx in 5C & Rx in 499B	20	3.40	34	56
Tx in 5B & Rx in 499C	21	3.00	26	41
Through 2 floors: Tx in 3D & Rx in 599G	40	2.97	93	30

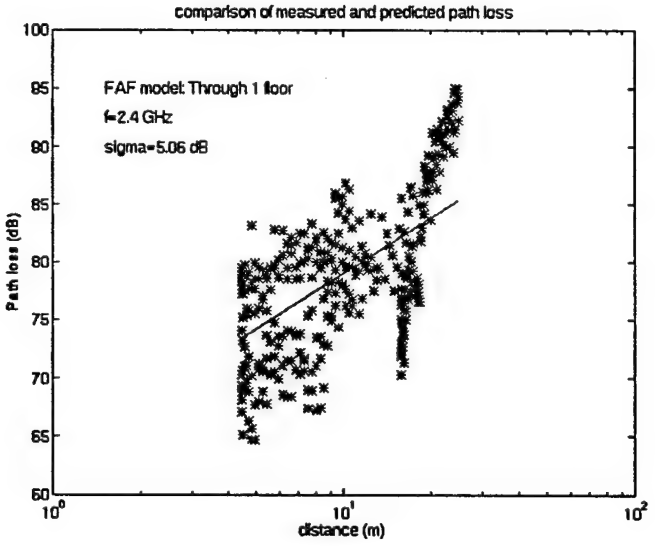
Table II gives the floor attenuation factors, the standard deviation (in decibels) of the difference between the mean path loss measured and predicted path loss, the mean RMS. Delay spread (in nanoseconds) and the number of discrete measurement locations used to compute the statistics. The same floor exponent (determined above with LOS) is found to be 1.6. One can notice that the FAF is not a linear function of the number of floors between the transmitter and receiver.

For propagation through 1 floor, the average FAF is found to be 23 dB and 40 dB for propagation through 2 floors. As shown in [1], different floors cause different amount of path loss. Many factors, including multiple reflections from surrounding materials near the transmitter and receiver, affect the path loss.

It is unclear what causes the differences between the FAFs for the propagation from the fifth to the fourth floor and the propagation from the third to the fourth floor. There may be other factors relative to the ceiling or the roof of the building since the fifth floor is the top one of the new engineering building of the UF.

Moreover, work in [2] showed that for indoor propagation, the reciprocity T-R is not always observed. In fact, in free space, propagation losses are identical if one interchanges receiver and transmitter locations while in indoor, each transmitter and receiver surrounding environment influences differently the propagation losses.

However, considering the RMS delay spreads, we can assume the received signal from the third floor is the result of more reflections which result in greater path loss.



(3) Path loss for propagation through 1 floor

C. Wall attenuation factor model

This model will include site specifications in addition to the T-R separation and number of floors taken in account in the previous models. That will lead to a more accurate propagation prediction.

For indoor propagation, there are often obstructions between the transmitter and receiver even when the terminals are on the same floor. We consider the path loss effects of concrete walls and soft obstructions (i.e. office furniture, bookcase). For this model, we assume path loss increases with distance as in propagation with LOS (n = 1.6) as long as

there are no obstructions between the transmitter and receiver. Then we include attenuation factors for each soft obstruction and concrete wall that lie between transmitter and receiver. In the model developed in [1], the authors assumed the propagation in building without obstacles behaves like a free space propagation taking the $n = 2$. However, we think that propagation in buildings without obstructions is rather equivalent to propagation with LOS. That is why we are considering using the LOS result of $n = 1.6$.

The mean path loss predicted by this model is then given by:

$$\overline{PL(d)}_{[dB]} = \overline{PL(do)}_{[dB]} + 10 \times n \times \log_{10}\left(\frac{d}{do}\right) + p \times AF_{wall} + q \times AF_{soft}$$

Equation of model (3)

where:

- ***p*** is the number of concrete walls between Tx and Rx
- ***q*** is the number of soft obstructions between Tx and Rx
- ***d***, ***do*** and ***PL(do)*** are as defined above
- ***AF*** is the attenuation factor

For each measurement, we recorded the number of soft obstructions(i.e. office furniture: bookcase, ...) and concrete walls that lie between the transmitter and receiver then we computed the difference between the path loss measured and the predicted path loss. We also used a linear regression in a MMSE sense to find the best fit to (3).

Table III

Locations	***AFsoft***	***AFwall***	***σ[dB]***	***Mean RMS delay spread (ns)***	***Number of measurements***
Through 1 wall: All locations		5.5	2.68	29	204
Tx in 4A & Rx in 408		6	2.21	NA	40
Tx in A (409) & Rx in 499C		5	2.45	27	54
Tx in B (409) & Rx in 499C		4	3.61	27	54
Tx in A (499F) & Rx in 470		6	1.84	27	28
Tx in B (499F) & Rx in 470		6.5	2.19	33	28

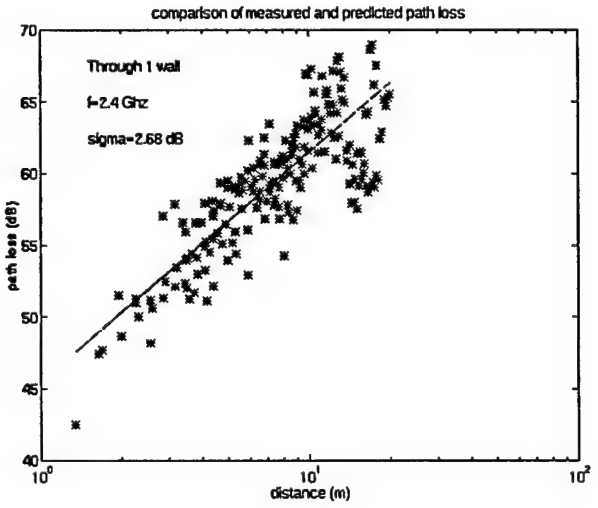
Through 2 walls: All locations	0.30	6	2.86	39	97
Tx in 436 & Rx in 499A	0.30	5	2.96	NA	36
Tx in 409 & Rx in 408	0.30	6.5	1.82	40	32
Tx in 409 & Rx in 403	0.15	5	2.82	38	29

Table III summarizes the mean path loss attenuation factor for soft obstructions and concrete walls, the standard deviation (in decibels) relative to the model, the mean RMS delay spread and the number of measurements in different locations.

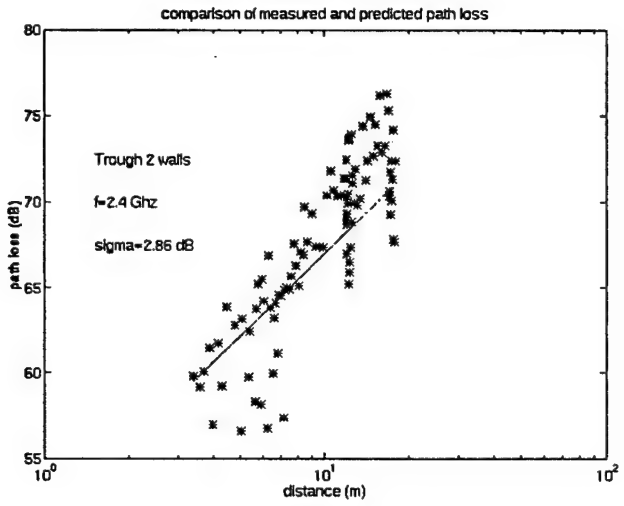
From this table, we can see that the mean AF (soft obstructions) was found to be as small as ***0.30 dB*** and the mean AF (concrete walls) to be ***5.5 dB***.

When each soft obstructed environment is considered separately, the attenuation factors range from 0.15 to 0.30 dB according to the amount of books in each bookcase or whether it is a board or not. In this case, our model is accurate to within 5 dB for as much as ***91%*** of measurements. The attenuation factors for soft obstructions found here correspond to the results in [3] where it has been shown that nonmetallic furniture had little effect on attenuation.

The absence of notable high reflecting obstacles has resulted in a mean RMS delay spread as low as ***38 ns*** for propagation through 2 walls.



(4) Path loss for propagation through 1 wall



(5) Path loss for propagation through 2 walls

4. Conclusion

Different path loss models have been presented for propagation at 2.4 GHz in the new engineering building of the University of Florida. The models are based on a simple algebraic relationship of the type ***PL=a+10*n*log (d)***. For propagation without any obstacles between the transmitter and receiver, this exponent is found to be 1.6 and is still close to 2 (the free space path loss exponent). That is a result of the effects of multiple reflections inside building. The floor attenuation factor model is appropriate to predict the effects of the number of floors between the transmitter and receiver.

When additional site specifications (i.e. number of concrete walls, soft obstructions between the transmitter and receiver) are used in the attenuation factor model, we can get a more accurate path loss prediction of indoor propagation with obstacles.

The resulting standard deviations prove the models to be accurate for ***82%*** of the LOS measurements, ***62%*** of the floored attenuated propagation measurements and ***92%*** of the obstructed propagation measurements to within ***5 dB***. Thus, it seems the method used above can offer an improvement in indoor communication system design and installation.

One method of characterizing wide-band multipath channels is by calculating their RMS delay spread ***(σ_d)***. For indoor propagation, when there are high reflecting obstacles in the transmitter surrounding environment, it is likely the RMS delay spread increases. That has been observed in several of the above measurements. Even if Fung and Rappaport have shown that delay spread alone does not determine the actual bit error rate at any instant, it does indicate the length of the channel matched filter and the necessary modulation dependent architecture.

As Theodore S. Rappaport said:" Models that allow a system designer to predict path loss

contours for all types of buildings without measurements would be extremely cost-effective and time-efficient." The influence of different building materials and the great variability in architectural configurations limit the accuracy of any model and its applicability to a prediction method for signal attenuation within buildings. That's why, more and more measurements are needed from different buildings and environments and at different frequencies in order to develop these models.

5. Summary

A. Propagation with line of sight (LOS)

$$\overline{PL}(d)_{[dB]} = \overline{PL}(do)_{[dB]} + 10 \times n \times \log_{10}\left(\frac{d}{do}\right)$$

With do=1 m it comes,

$$\overline{PL}(d)_{[dB]} = 40 + 16 \times \log_{10}(d)$$

$$\sigma_d = 60ns$$

$$\sigma_{PL} = 3.75dB$$

$$probability = 82\%$$

(within 5 dB)

B. Propagation through floors

$$\overline{PL}(d)_{[dB]} = \overline{PL}(do)_{dB} + 10 \times n \times \log_{10}(d) + FAF$$

1) Propagation through 1 floor:

$$\overline{PL}(d)_{[dB]} = 16 \times \log_{10}(d) + 63$$

$$\sigma_d = 37ns$$

$$\sigma_{PL} = 5.06dB$$

$$probability = 62\%$$

(within 5 dB)

2) Propagation through 2 floors:

$$\overline{PL}(d)_{[dB]} = 16 \times \log_{10}(d) + 80$$
$$\sigma_d = 93ns$$
$$\sigma_{PL} = 2.97dB$$
$$probability = 93\%$$

(within 5 dB)

C. Propagation through obstacles (on the same floor)

$$\overline{PL}(d)_{[dB]} = \overline{PL}(do)_{[dB]} + 10 \times n \times \log_{10}(d) + p \times AF_{wall} + q \times AF_{soft}$$

$$\overline{PL}(d)_{[dB]} = 40 + 16 \times \log_{10}(d) + p \times 5.5 + q \times 0.30$$
$$\sigma_{d1} = 29ns$$
$$\sigma_{d2} = 39ns$$
$$\sigma_{PL} = 2.7dB$$
$$probability = 92\%$$

(within 5 dB)

Where:

- p is the number of concrete walls
- q is the number of soft obstructions (furniture: bookcase, ...)

The standard deviations are obtained with the transmitter in rooms or hidden from metallic doors.

SPREAD SPECTRUM TECHNOLOGIES

The WLANs often use an unlicensed frequency like the 2.4 GHz in the ISM band. Hence, to avoid interference with other systems, different alternatives have been required.

One possibility is to simply use a very low transmitting power but this is not possible for most of the applications that require more speed and more range. The other possibility is to use a Spread Spectrum (SS) technology. Two different SS technologies already exist and are:

- The DSSS (Direct Sequence Spread Spectrum)
- The FHSS (Frequency Hoping Spread Spectrum)

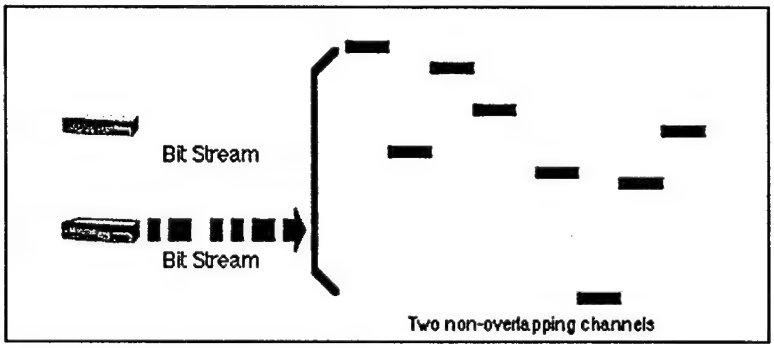
In both technologies, the goal is the same: spreading the transmitting power into a larger band instead of concentrating it in a single narrow frequency band.

1. FHSS

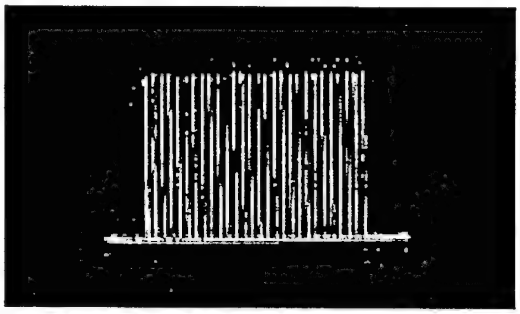
FHSS principle is to transmit a short burst of the signal on one frequency, hopping to another frequency for another short burst and so on. Both the transmitter and the receiver must be synchronized in order to be on the same frequency at the same time.

Moreover, the hopping pattern (frequency and order in which they are used) and dwell time (time at each frequency) must be known by the source and the destination of the signal. The Federal Communications Commission (FCC) requires at least 75 frequencies and a dwell time of 400 ms. If interference occurs on one frequency, the data is retransmitted on a subsequent hop on another frequency.

By using different orthogonal hoping sequences, more than one channel can be used at the same time.



(6) FHSS illustration



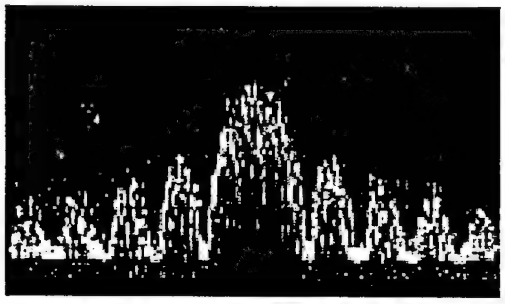
(7) FHSS Spectrum

2. DSSS

The principle is to increase the bandwidth while mapping each bit into a pattern of chips: the spreading sequence. At the receiver, the chips are mapped back into bits to recreate the original data. Synchronization between the transmitter and the receiver is also required.

The ratio of chips per bit is called the "spreading ratio". A high spreading ratio increases the resistance of the signal to narrowband interference. A low spreading ratio increases the net bandwidth available to a user.

In practice, spreading ratios are quite small - often less than 20. The proposed IEEE 802.11 standard specifies a spreading ratio of at least 11. The FCC just requires that the spreading ratio must be greater than 10.



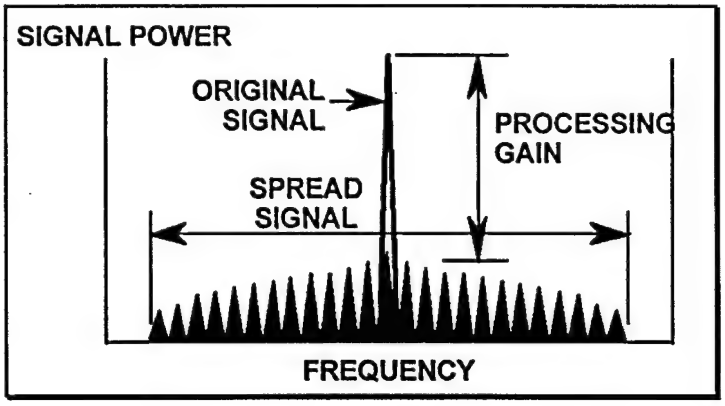
(8) DSSS Spectrum

3. Processing gain

Processing gain (PG) is a term used to describe one of the unique properties of Spread Spectrum waveforms. It helps to measure the performance advantage of spread spectrum against narrowband waveforms.

The PG for a FH systems is defined as the ratio between the instantaneous bandwidth of each hop and the overall bandwidth of the transmission channel in dB. A DS system defines the PG as the ratio between the chip rate and the bit rate in dB.Higher PGs in both SS systems help reduce the amount interference the system will receive.

The PG is easy to see with a DS spectrum:



(9) Illustration of the DSSS Processing Gain

4. Comparison between both SS technologies

Those two technologies have the same goals but not the same principles. They were used during a long time only for military applications in order to reduce the probability that the communications may be understood by the enemy. Now, most of the wireless communication systems makers try to use these technologies in civil applications. Hence, it is quite hard to get an objective idea of what is the best technology.

Here are a few arguments found in different sources that present the main differences.

- The frequency hopper is more difficult to synchronize because both the time and frequency need to be in tune. While in a direct sequence radio, only the timing of the chips needs to be synchronized. The frequency hopper will need to spend more time to search the signal and lock to it. Therefore, the latency time is usually longer, while a direct sequence radio can lock in the chip sequence in just a few bits.

- The frequency hopping technique does not spread the signal, as the DSSS does. The processing gain of a DSSS system is the increase in power density when the signal is despread and it will improve the received signal's S/N ratio. In other words, the frequency hopper needs to put out more power in order to have the same S/N as a direct sequence radio.

- The frequency hopper, however, is better than the direct sequence radio when dealing with multipath. This is because the hopper does not stay at the same frequency and because a null at one frequency is usually not a null at another frequency if it is not too close to the original frequency. So a hopper can usually survive the multipath better than direct sequence radio.

- Frequency hopped signals will generally have better adjacent channel selectivity compared to DS spread signals. However, FH radios must hop through 50 channels. The ETSI requires this to keep spectrum usage uniform and random. Selective use of channels is not allowed in frequency hopping. DS radio users have the freedom of selecting the channels that have the least amount of traffic and interference in their area.

- DS spread radios also offer the opportunity for better power management than FH radios. A DS radio can more easily rely on the wireless network access points to determine when it can shut down to conserve power. FH systems are forced to stay on more of the time because of the need to constantly synchronize their hopping sequence with that of the RF network access points. Therefore, battery life is potentially longer with DS spread radios than it is with their FH counterparts.

HARRIS PRISM CHIPSET PRESENTATION

1. Introduction

PRISM is a range of components that Harris Corporation developed for portable wireless communications systems. The propagation frequency used is 2.4 GHz and one of the most interesting features of this range is the use of Direct Spectrum Spread Sequence (DSSS).

All the components designed under this name are highly integrated in order to allow miniaturized systems such as the following:

- Wireless Local Area Network (WLAN),
- Point-to-point microwave communications systems,
- Handheld data transceivers,
- Telemetry...

This range is packed into five interoperable ICs and but also a complete solution for WLAN within a PCMCIA PC card. This card is a complete wireless high-speed modem using the PRISM DSSS Wireless Transceiver chip set. This is the easiest way to design a simple WLAN, that is why it was the more interesting to simulate.

2. The PRISM PCMCIA card

A. Presentation

This card allows high bit rate until 2 MBPS and its specifications match the IEEE 802.11 Direct Sequence Specifications. Its propagation frequency is 2.4 GHz (unlicensed ISM band: Industrial/Science/Medical band).

Its main features are:

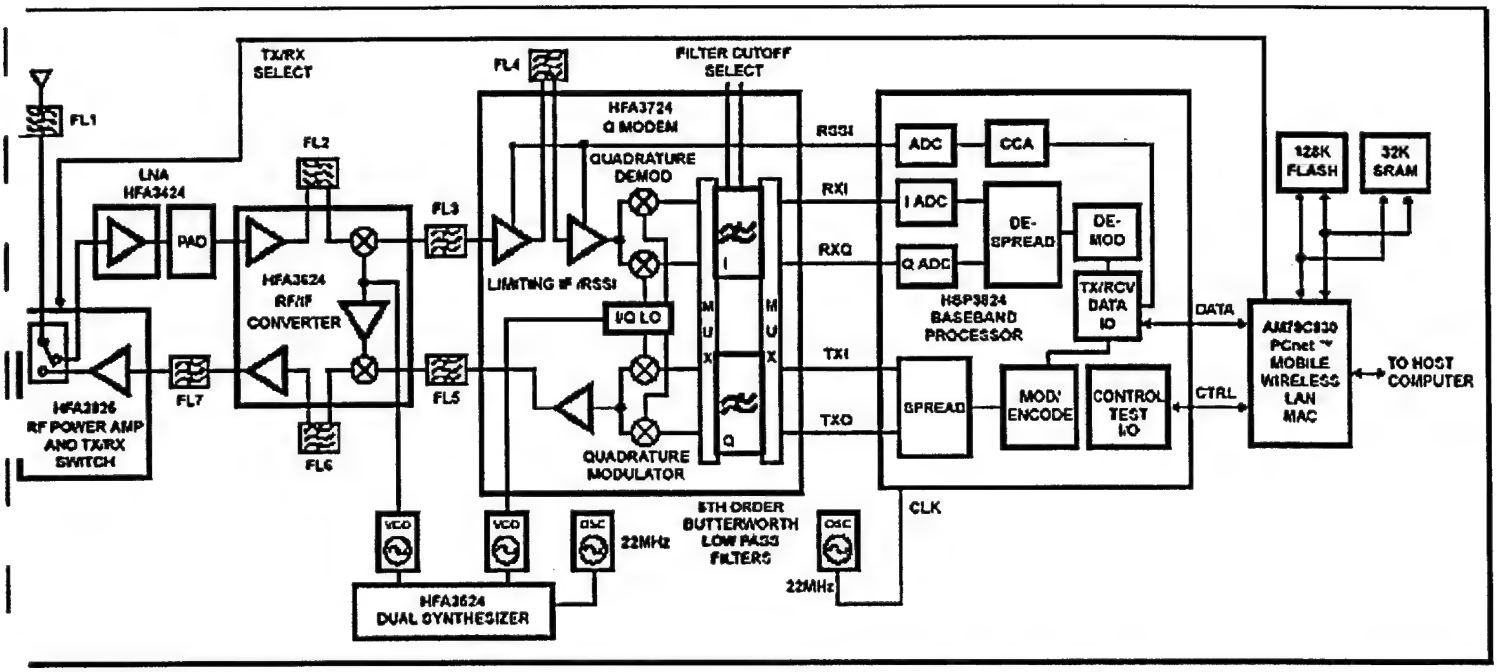
- Frequency range : 2.4 GHz - 2.4835 GHz
- Transmitter output power: 20.5 dBm (with the built-in antenna)
- IF Frequency (Intermediate Frequency) : 280 MHz
- Modulation type : DBPSK or DQPSK
- Binary rate : 2 MBPS in DQPSK and 1 MBPS in DBPSK
- Receiver sensitivity : -93 dBm at 1 MBPS and -90 dBm at 2 MBPS, both with a frame error rate of 8 %

- Half duplex transmission

This is packaged into a PCMCIA Type II extended cover including a 2.5 dB integrated microstrip antenna. Both Windows 3.x and Windows 95 support this WLAN solution.

The radio range depends on the environment and one of the goals of this simulator and propagation modeling is to predict it.

B. Architecture



(10) PRISM PCMCIA Card Synoptic

This card is designed around the following PRISM components:

- The HSP3824 baseband processor
- The HFA3724 Q-MODEM
- The HFA3524 dual synthesizer
- The HFA3624 RF/IF converter
- The HFA3925 RF power amplifier and TX/RX switch

The other main component is the Media Access controller (MAC) which is mainly in charge of the communications protocol within the WLAN. This is a PCnet AM79C930.

3) Transmit processing

Once the MAC has sent the data to the baseband processor and after the latter had encoded them (differentially coding), each bit is coded using the predetermined 11 chips per bit Spreading Sequence. These logical data are then sent to the MODEM in order to create an IF frequency (Intermediate Frequency) signal.

The MODEM modulates the data to an IF frequency of 280 MHz using BPSK or QPSK modulation. In order to get the IF frequency, an output of the dual synthesizer is applied to the LO input of the MODEM. As the MODEM divides the frequency by two, the frequency applied to the LO is 2 x 280 = 560 MHz. Hence, the second LO output of the dual synthesizer is chosen. The signal is then amplified and filtered before being transposed at 2.4 GHz via the IF/RF converter.

This last one mixes the input signal to the first output of the dual synthesizer to get the 2.4 GHz. Finally, the signal is amplified and filtered twice before entering the antenna. The transmitted power is about 20.5 dBm.

4) Receive processing

The processing is almost the same as the transmit processing except that the signal is amplified more significantly in order to recover the data whatever the conditions.

Following reception via antenna and then filtering, the signal is sent to a Low Noise Amplifier (LNA) in order to reduce the Noise Figure (NF) of the whole reception. However, in order to improve the input intercept point, a high attenuation may be required. A compromise was found using both the LNA and 5-dB attenuator.

The signal is then down-converted by the RF/IF converter and filtered before being sent to the input of the MODEM. First, the signal is limited twice with two 45-dB limiting amplifiers in order to recover an adequate amount of power. The demodulation is then achieved like the modulation in the transmit processing.

At this moment, the signal is still analog and the baseband processor has to sample it in order to recover the binary sequence. This is made at twice the chip frequency (generally 44 MHz = 2 x 2 MBPS x 11 chips per bit). Then, the baseband processor correlates the PN spreading (Spread Sequence) to recover the differential binary data. These data are then decoded (differential to absolute values) before being sent to the MAC.

A transmit and receive processing detailed explanation is given in the annex in the Harris Application Note 9624.

THE SIMULATION

1. Features and goals

The simulator is used to get a precise idea of what happens in all stages of the transmit and receive processing. One of the first purposes is to let the user know exactly the shape of the signal at each stage. In that sense, it is a pedagogic tool because the user can see the importance of each component and may better understand why such an amplifier or filter was chosen instead of another.

Then, the simulator allows us to predict the quality of the received signal depending of the hardware and the channel characteristics (cf. Modeling part). Almost all the parameters of the transmission can be chosen and even changed by the user who can get an idea of the best way to use the transceivers: which kind of modulations, which spreading sequence and so on...

The simulator takes into account each filter's transfer function, the gain, attenuation and noise figure for each component and even the power of the signal at each stage. All of the fundamental parameters such as the IF and RF frequency, the bit rate, the sampling frequency, the constellation of the modulation, and the spreading sequences have been carefully detailed.

The simulation was designed under Matlab not only because of its very powerful mathematical functions but also because it's a current software and easy to use under both MSDOS/Windows and UNIX operating systems. Moreover, the main parameters can be simply chosen via a convivial user interface.

2. Program structure

The program is divided into 7 M-scripts and 7 functions in order to get an easy-to-understand program.

The description of each of those files is given below:

- *Run_i.m* is the routine used to create the user interface.
- *Test.m* is the subroutine called by *Run_i.m*. This describes the callback actions that match the buttons pushed by the user.
- *Mainpris.m* is the program used to choose all the main parameters of the simulation. It is called by *test.m* after the user choices have been done. Some of those parameters are decided via the user interface but the others can simply be changed in this script. This is the main script which is used to call the transmitter, the channel and then the receiver routines. It also loads the filters taps defined in the *filtrage.m* script. All the parameters are then saved into the *Variables/globalva.mat* file.

- *Emit.m* is the script that matches the transmit processing. First, it creates a random binary sequence using the *gene_bin.m* function. Then, this sequence is used to create a symbol sequence with *gene_sym.m* - each symbol has a length of two bits for a QPSK modulation and a length of just one bit for a BPSK modulation. At that point, the *mapqam.m* function creates the PSK Q and I signals that will be differentially coded.

  Next, both signals are correlated with the spreading sequence using the *spread.m* function. After a first filtering, the signals are mixed up with a 280 MHz LO (Local Oscillator) and then added together to create the IF modulated signal.

  For the numerous filtering stages, the *filtre_np.m* function used to be called to calculate the result of the filtering on the signal and on a pure signal without any noise. This second signal is used as a reference to calculate the noise power of the real signal which is the last output of this function.

  When the signal meets an amplifier, two different functions can be called. The first one *gain_nf.m* calculates the result signal while taking in account both the gain and the Noise Figure (NF) of the signal. The second one *gain.m* just takes in account the gain of the amplifier and is just used to amplify the non-noisy reference signal.

  The signal is of course also mixed to be up-converted at 2.4 GHz and the output of this script is the radio-transmitted signal after the 2.5 dB antenna.

- *Channel.m* is the second main script used to take in account the result of the modeling (Cf. Modeling part) to attenuate the transmitting signal and add it -100 dBm of noise which corresponds to the noise in the air.

- *Receiver.m* is the biggest script. It matches all the receive processing but is also used to display a comparison between the transmitted signals (binary sequence, I and Q signals) and the received ones. It uses the same routines as *emit.m* except the first one's used to create the binary sequences.

- *Gene_bin.m* is used in *emit.m* to create a random binary sequence. It is a function whose syntax is:

  binary_sequence=gene_bin(N,p0) where *binary_sequence* is the output binary sequence obtained randomly, *N* its length and *p0* the probability to get a zero.

- *Gene_sym.m* is used in *emit.m* to create a random binary sequence. It is a function whose syntax is:

  symb_sequence=gene_sym(binary_sequence,M) where *symb_sequence* is the output symbol sequence obtained using the input *binary_sequence*. *M* is the number of different symbols. *M* is 2 for a BPSK and 4 for a QPSK.

- *MapQAM.m* is used in *emit.m* to create the I and Q signals that match the chosen constellation and modulation. The syntax is:

  [DI,DQ]=mapQAM(symb_sequence,M) where *DI* and *DQ* are the output I and Q signals (Q=0 for a BPSK), *symb_sequence* is the input symbol sequence calculated with the previous function and *M* is the number of different symbols.

- *Filtrage.m* is a M-script used to calculate all the taps of the different filters used in both

the transmit and the receive processing. Then, they are saved into the *Variables/globalva.mat* file.

- *Spread.m* is a function used to apply a pre-determined spreading sequence to both I and Q signals. The syntax is:

  [SI,SQ]=Spread(DI,DQ,SS,fs,fbr,N,delay) where *SI* and *SQ* are the output spread signals and *DI* and *DQ* the input. *Fs* is the sampling frequency, *fbr* the binary rate, *N* the number of points and a *delay* (number of samples) can be set before the spread sequence is applied to the input signals (useful for the receive processing after the numerous digital filters).

- *Gain_nf.m* is used to amplify a signal while taking in account the noise figure of the component. Its syntax is:

  [output_sig, out_noise_pwr] = gain_nf (input_sig, dB_gain,dB_NF, inp_noise_pwr) where *output_sig* is the amplified output signal with a noise increased to match the NF. *Out_noise_pwr* is the noise power of the output signal, *input_sig* is the input signal, *dB_gain* is the gain in dB, *dB_NF* is the NF in dB and *inp_noise_pwr* is the noise power of the input signal.

  All noise powers are given in Watts. The gain is a power gain.

- *Gain.m* is used to amplify a signal with a power gain. The syntax is:

  output_sig=gain(input_sig,dB_gain) where *output_sig* and *input_sig* are the output and the input signals respectively, *dB_gain* is the power gain in dB applied to the input signal.

- *Filtre_np.m* is used to filter both a noisy signal and a non-noisy equivalent signal and then to get the noise power of the output signal while comparing both output signals. The syntax is:

  [noisy_sig_out,nn_sig_out,noise_pwr_out]=filtre_np(B,A,signal_in,signal_nn_in) where *noisy_sig_out, nn_sig_out, signal_in* and *signal_nn_in* are the output noisy signal, the non-noisy filtered signal, the input noisy signal and the input non-noisy equivalent input signal respectively. *B* and *A* are the upper and lower filter taps (for an IIR filter), respectively. *Noise_pwr_out* is the noise power of the noisy filtered signal.

3. Noise calculation

As the simulator is able to compute all the noise figures of all the components, we must know exactly everytime what is the noise power of the signal. Indeed, we just know the Noise figure (NF) of each component and:

$$NF = \frac{(S/N)_I}{(S/N)_O} = \frac{S_I}{S_O}\cdot\frac{N_O}{N_I} = \frac{1}{G}\cdot\frac{N_O}{N_I} = \frac{1}{G}\cdot\left(\frac{G.N_I + N_A}{N_I}\right) = 1 + \frac{N_A}{G.N_I}$$

where N_I, N_O, S_I,S_O and G are the input and output noise power, the input and output

signal power and the linear power gain respectively. N_A is an additive noise due to the component, which will be used to match the noise figure.

To create a noisy signal, at the beginning of the *emit.m* script, a random signal with a power that corresponds to the noise power was added. Then, everytime the signal pass through an amplifier, the new noise power is calculated using:

$$N_O = G.N_I + N_A$$

The noisy signal is doubled by the equivalent signal but without any noise. Each time a filter appears, both signals are filtered so that to get the new noise power using this formula:

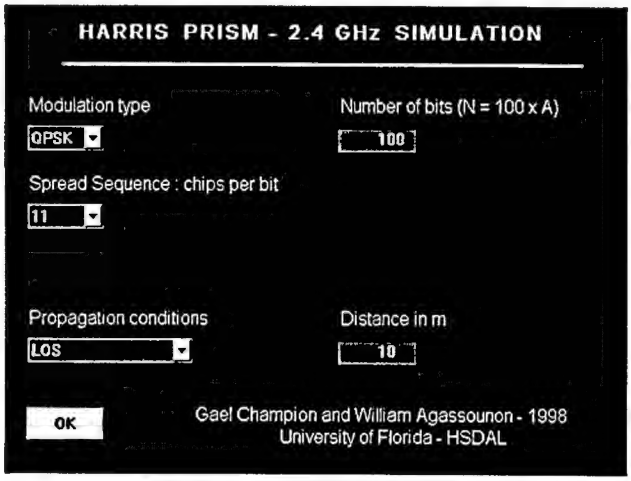
$$N_O = \frac{1}{T} \cdot \int_0^T \left[S(t) - Snn(t)\right]^2 dt$$

Where *S(t)* is the noisy filtered signal and *Snn(t)* the non-noisy corresponding signal.

As the whole program is under a loop that enables the user to choose the number of points he wants to be processed, the filtering of the non-noisy signal is only made once (when the first 100 bits are processed). Then, the noise power after each filtering is saved under a reference variable such as *ref_5*. This reference will be used directly in the computations for the noise power.

4. The user interface and the configurable parameters

When the program starts, a window appears to allow the user to set the main parameters of the transmission. Those can be simply changed using the mouse and clicking the few popup menus, the editable windows and then clicking OK to run the simulation.



(11) View of the window user interface

- The *Modulation type* is a popup menu use to select either a BPSK modulation or a QPSK modulation. The default option is QPSK

- The *Number of bits* is an editable text window used to fix the length in bits of the binary sequence. You must choose an integer that is a multiple of 100. The default option is 100.

- The length of the *Spread Sequence* in *chips per bit* can be selected with a popup menu. The possible choices are 11,13, 15 or 16. The default option is 11.

- The *Propagation conditions* are used to select the kind of propagation between the transmitter and the receiver. The choices are LOS (line of sight), propagation through one floor, through 2 floors, or through one or two walls on the same floor. The default option is LOS.

- The *Distance in m* is an editable text window used to select the distance (in meters) between the transmitter and the receiver. The default option is 10 m.

- The *OK* push button is just used to validate the user choice and to run the simulation.

Many other parameters can also be changed directly in the *Mainpris.m* M-script. Those parameters are non-directly related to the setting parameters of the PRISM PCMCIA cards but enable the user to test many other transmissions properties like:

- *Fc*, which is the intermediate frequency. The PRISM components used to support a frequency from 10 to 400 MHz. Default is 280 MHz

- *Fp*, which is the propagation frequency. The user must be notified that the propagation modeling has been done only for 2.4 GHz propagation. Hence, changing this parameter will not match the channel modeling. The results should then be wrong.

- *SS*, which is the 16-chip or the 11-chip spreading sequence. A 13 or 15-chip sequence is simply chosen by ignoring the last chips of the 16-chip sequence. Those sequences can be changed easily. The 11-chip and the 13-chip sequence correspond to a 11-bit and a 13-bit Barker code, respectively. If the length of the sequence is 11, the default code is 05B8. If it is 13,15 or 16, the code is 1F35. These two sequences have been chosen to give the best results.

- *Dividetap*, which is the divide tap used to fix the binary rate. The binary rate is calculated using this formula:

$$binary\ rate = \frac{clock\ frequency}{length(SS).Divide\ tap}$$

  Default is 2.

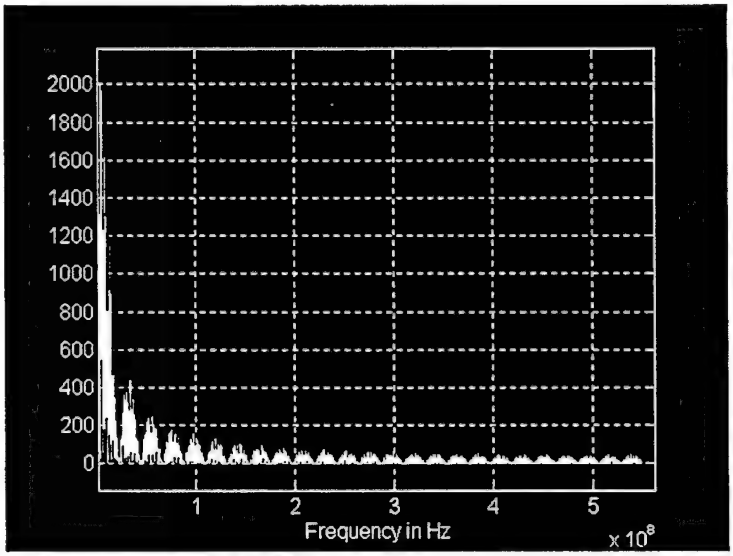
- *Vcc*, which is the voltage used to represent a digital 1. Default is 3 Volts.

- *Att_air*, which is used to set the noise floor in the air. Typically between -100 and -110 dBm. Default is -100 dBm.

5. Running the simulation

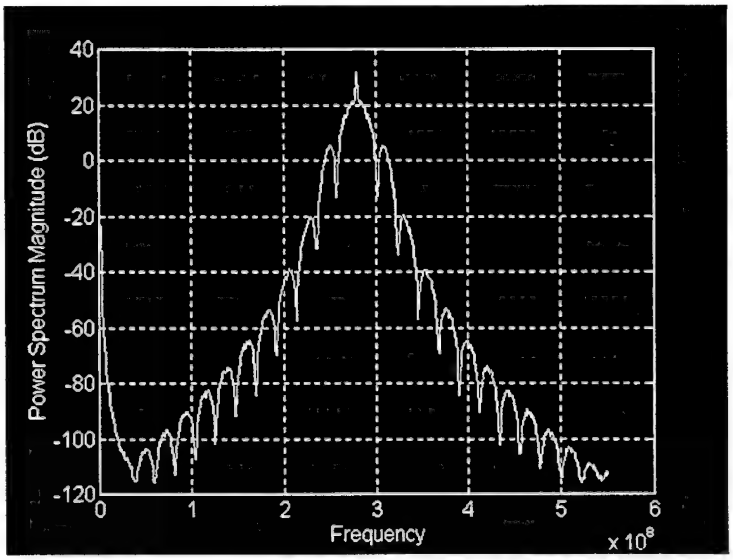
A. Displayed figures

During the simulation, many figures (17) appear to present the signal at different stages of the transmit and receive processing. The goal is to help the user figure out the purpose of each component such as amplifiers or filters based on its spectrum. After receive processing, the processed signal is compared to the transmitted sequence. Finally, the received binary sequence is compared to the transmitted one and the familiar bit error rate versus signal to noise ratio graph is displayed. The figures correspond to a 10000 Monte Carlo simulations of 100 bits.



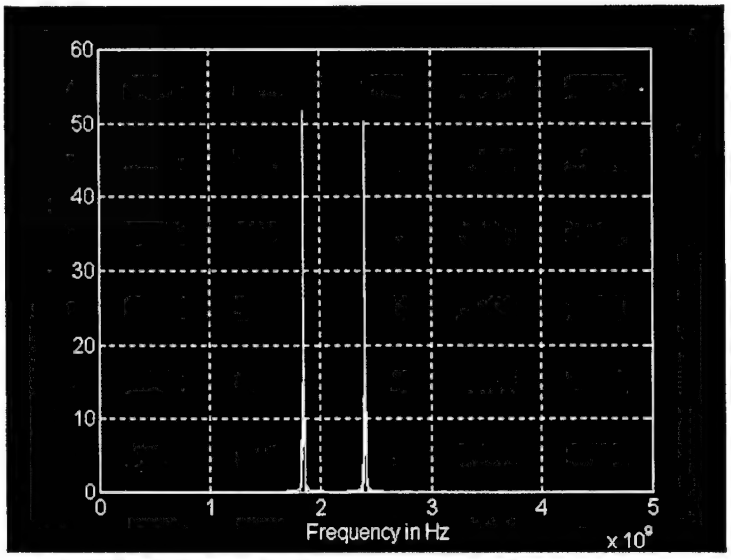
(12) SI Spectrum after DSSS

The figure 12 (number 1 in the simulation) shows the goal of the DSSS which is to increase the bandwidth of the signal. The binary rate is 2 MBPS (default value) but the width of each lobe is extended to 22 MHz (44 MHz for the first main lobe) with the 11-chips per bit spreading sequence.



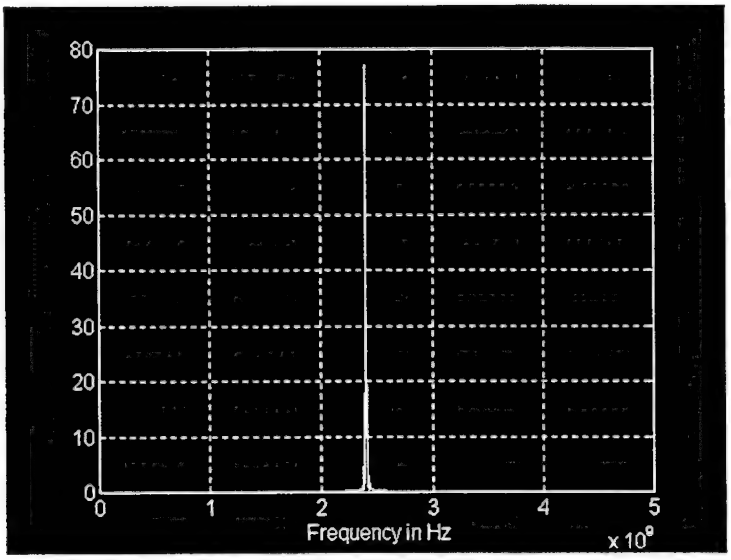
(13) IF Spectrum before filtering

The figure 13 (number 2 in the simulation) shows the same spectrum but after the modulation at the IF frequency. The result is the same but the spectrum is centered on 280 MHz, which was the IF frequency chosen.



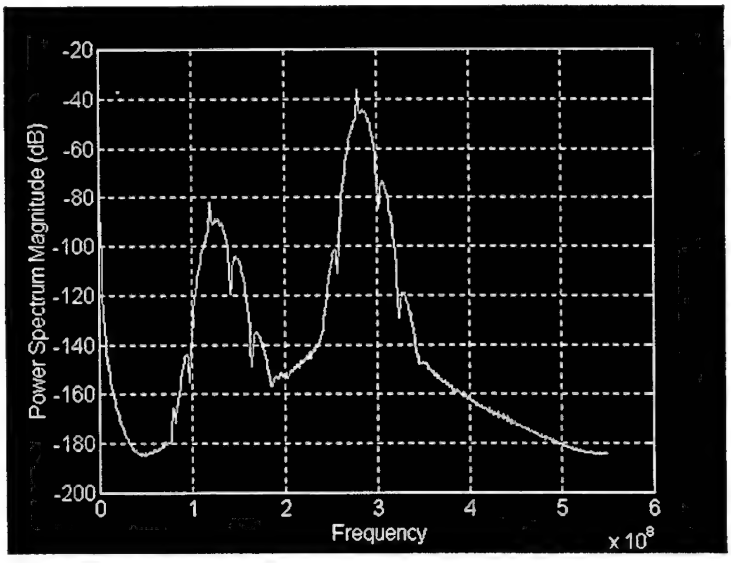
(14) RF FFT before FL6 filter

The figure 14 (number 4 in the simulation) shows the spectrum of the transmitted signal after the RF mixing. The 280 MHz signal was mixed with a 2.12 GHz LO in order to transpose the signal at 2.12 + 0.28 = 2.4 GHz. However, during the mixing, another frequency is generated: 2.12 - 0.28 = 1.84 GHz. That is what appears on this figure.



(15) RF FFT after FL7 and FL6 filtering

The figure 15 (number 6 in the simulation) shows the same spectrum but after FL6 and FL7 filters whose goal is to remove the undesired frequency (1.84 GHz). The signal has also been amplified and hence its amplitude is higher.



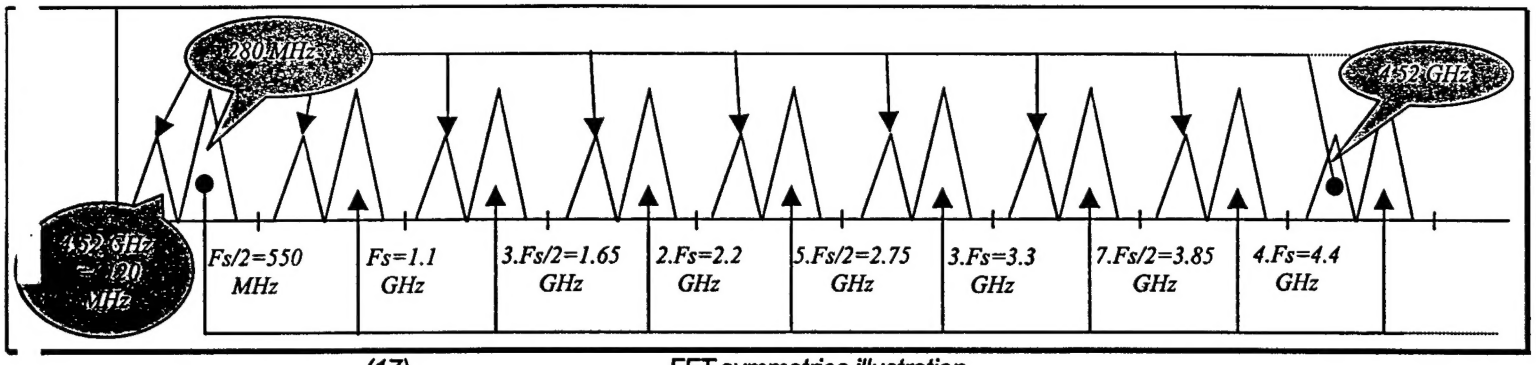
(16) Received IF Spectrum after FL3

The figure 16 (number 11 in the simulation) shows the spectrum of the received signal mixed with a 2.12 GHz LO to recover the signal at the IF frequency (280 MHz). The main lobe centered at 280 MHz is comparable to the transmitted IF spectrum.

Nevertheless, another frequency seems to appear at 120 MHz. This is due to the other

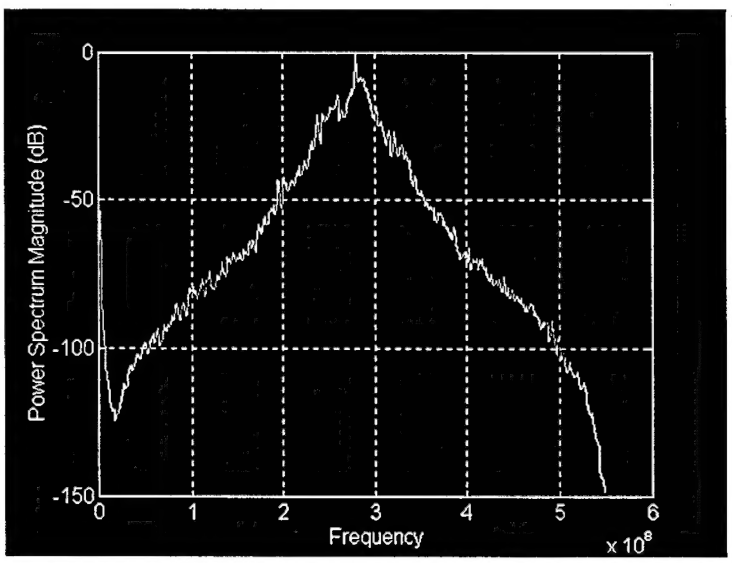
frequency generated by the down conversion. Indeed, a 2.4 + 2.12 = 4.520 GHz frequency is generated by the mixing. Indeed, the spectrum is calculated using an FFT and an FFT is symmetric to every multiples of Fs/2 and gets repeated every Fs, where Fs is the sampling frequency.

This is illustrated by the following picture.



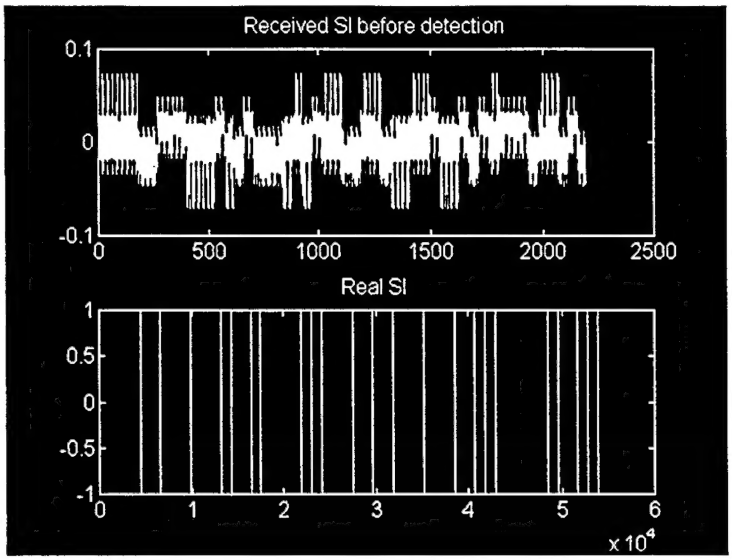
(17) FFT symmetries illustration

However, this undesired frequency of 4.52 GHz is attenuated by the use of the FL3 filter.



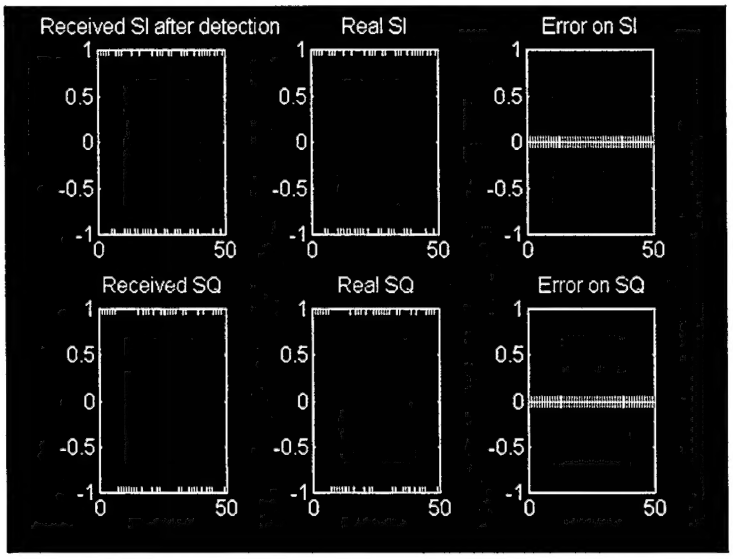
(18) Received IF Spectrum after FL4

The figure 18 (number 12 in the simulation) shows the same spectrum but after both a limiting amplifier (which increases the power of the signal but also modifies a little bit the shape of the spectrum) and another filtering due to FL4. The 4.52 GHz frequency (represented at 120 MHz) is completely removed.



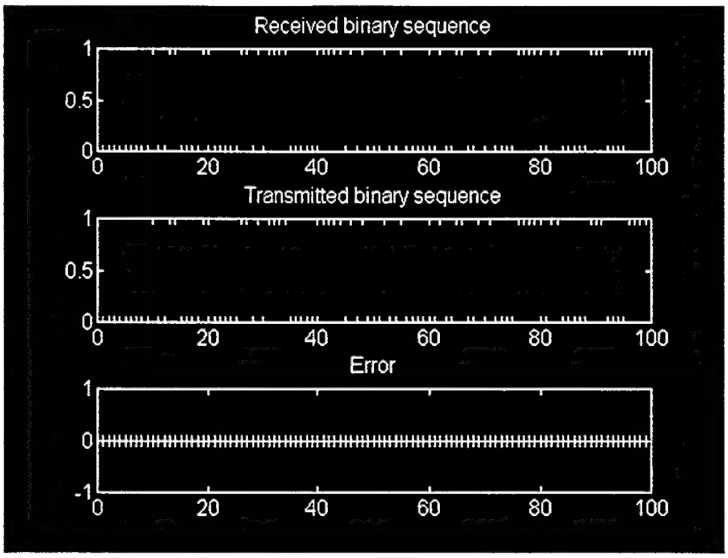
(19) Comparison between transmitted and received SI before detection

The figure 19 (number 14 in the simulation) is just used to get an idea of the received demodulated signal after despread but before detection. The figure number 15 of the simulation is the same but for SQ.



(20) Comparison between transmitted and detected received SI and SQ

The figure 20 (number 16 in the simulation) is a comparison between the detected SI and SQ and the transmitted SI and SQ. An error graph for both signals is also displayed.



(21) Comparison between the transmitted and received binary sequences

The figure 21 (number 17 in the simulation) shows the transmitted and the received binary sequence. A graph error is also displayed.

B. Results

As previously said, once the simulation is over, the binary error rate is displayed. This is in the case that the reception of the signal is possible.

Indeed, the limiting amplifiers used in the HFA 3724 Q-MODEM cannot work with signals whose power is below -84 dBm. This corresponds to the sensibility of this component. In that case, the limiting amplifier will not be able to work and so no signal will be detected.

Hence, the signal power is calculated in the receiver simulation at the stage that corresponds to this component. If the signal is not powerful enough, the simulation simply stops while displaying this message:

'RECOVERING IMPOSSIBLE - THE SIGNAL IS NOT POWERFUL ENOUGH'

In both cases, the simulation stops while displaying this end message:

'End of simulation

Use the Windows menu to view the different figures'

A sound is also emitted, as the simulation can be quite long. Indeed, it requires more than ten minutes to compute a binary sequence of 500 bits with the default options on a Pentium II 300 MHz with 64 MB of RAM.

CONCLUSION

A wireless communication simulator and a statistical distance-dependent path loss prediction model are useful for understanding the propagation of radio waves in Wireless Local Area Networks (WLANs).

The simulator was designed to keep as close as possible to the whole Harris PRISM PCMCIA Card technology. Transmit and receive processing are analyzed step by step so as to provide an accurate overall view and a practical tool when designing wireless communications links. It can also be of high interest for data transmission analysis when the Bit Error Rate becomes a critical matter.

In addition to this qualitative work, a statistical propagation path loss modeling gives both a theoretical and practical overall view of the influences of indoor channels according to site specifications such as floors, walls, soft obstructions, etc.

The integration of both the simulator and the modeling results into an easy-to-use Matlab program with a convivial user interface gives a complete solution in order to get an accurate idea of the real features of your next WLAN.